Skriptum Informatik

- eine konventionelle Einführung

Prof. Dr. Hans-Jürgen Appelrath
Universität Oldenburg

Prof. Dr. Jochen Ludewig
Universität Stuttgart

Verlag der Fachvereine Zürich

B. G. Teubner Stuttgart

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Appelrath, Hans-Jürgen:

Skriptum Informatik : eine konventionelle Einführung / Hans-Jürgen Appelrath ; Jochen Ludewig. – Zürich : Verl. der Fachvereine ; Stuttgart : Teubner, 1991

ISBN 3-519-02153-6 (Teubner)

ISBN 3-7281-1799-4 (Verl. der Fachvereine)

NE: Ludewig, Jochen:

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung der Verlage unzulässig und strafbar. Das gilt besonders für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

© 1991 B. G. Teubner, Stuttgart

und Verlag der Fachvereine an den schweizerischen Hochschulen und Techniken, Zürich

Printed in Germany

Druck und Bindung: Präzis-Druck GmbH, Karlsruhe

Umschlaggestaltung: Fred Gächter, Oberegg, Schweiz

Vorwort

Der Titel dieses Buches ist Beschreibung und Abgrenzung: Wir legen ein Skriptum einer zweisemestrigen Einführungsvorlesung Informatik vor, also einen Text, der im wesentlichen anlässlich von Lehrveranstaltungen der Verfasser an der ETH Zürich (1985-88), der Universität Stuttgart (1988-90) und der Universität Oldenburg (1989-90) aufgeschrieben wurde. Und diese Vorlesungen für Studenten¹ der Hauptfach-, einer Nebenfach- oder einer sogenannten Bindestrich-Informatik waren und sind *konventionell* in dem Sinne, daß das Erlernen einer modernen, aber imperativen Programmiersprache (MODULA-2) ein wichtiges Teilziel darstellt. Auch das Vorgehen dabei, bottom-up, vom kleinen zum großen, ist durchaus konventionell.

Die Wahl der Sprache ist umstritten; andere Ansätze legen eine nicht-imperative Sprache zugrunde oder ziehen es vor, die Theorie besonders zu betonen. Jeder kann für sein Konzept gute Gründe vorbringen; unsere sind die folgenden:

- In der Praxis - und dort werden fast alle nach erfolgreichem oder auch abgebrochenem Studium landen - haben die nicht-konventionellen Programmiersprachen bis heute keinen nennenswerten Anteil erreicht; daran wird sich auch in der absehbaren Zukunft nicht viel ändern. Nur durch die Ausbildung mit einer imperativen Sprache können wir also einen Beitrag zu besserem *Software Engineering* leisten. Die Vermittlung der *Konzepte* genügt nicht, denn Programmieren hat auch eine starke handwerkliche Komponente, die detaillierte Hinweise und Übung erfordert.
- Kaum ein Student beginnt das Informatik-Studium ohne Programmiererfahrung. Leider sind diese Erfahrungen zum großen Teil nicht nützlich; noch immer geistert BASIC durch private und leider auch schulische Rechner, und auch was in Pascal gefaßt wurde, ist oft kein sauberer, systematisch entwickelter Code. Vor diesem Hintergrund mag dieses Skriptum auch als Orientierungshilfe für den Informatik-Unterricht in der Oberstufe dienen. Unterrichten wir die Erstsemester nur in Prolog oder in einer anderen nicht-konventionellen Sprache, so setzen wir dieser Fehlentwicklung nichts entgegen, sondern bauen eine entrückte *Zweitwelt* auf. Die erste im Grundstudium eingesetzte Sprache sollte praxistauglich und hinreichend verfügbar sein. MODULA-2 erfüllt beide Anforderungen.
- Eine Ausrichtung der Vorlesung auf den Objektbegriff erschien uns durchaus attraktiv, nur sehen wir keinen Weg, mit Objekten zu arbeiten, bevor die *elementaren* Begriffe systematisch eingeführt sind. Darum schätzen wir MODULA-2 als eine Sprache, die die *Vorbereitung* des Objektbegriffs unterstützt. Da man in der Pro-

¹Wir verzichten in diesem Buch auf jeden Versuch, geschlechtsneutrale Formulierungen zu erreichen, denn alle uns bisher bekannten Lösungsansätze dieses Problems erscheinen unbefriedigend. Wir stellen allerdings ausdrücklich fest, daß dieses Problem besteht und einer Lösung bedarf.

grammierung das Detail beherrschen muß, bevor man zum "Programmieren im Großen" übergeht, sehen wir auch keine Alternative zum bottom-up-Konzept.

- Nicht zuletzt: Wir haben mit MODULA-2 viel Erfahrung, so daß wir in der Vorlesung aus dem Vollen schöpfen können - und wir kennen die Unzulänglichkeiten gut genug, um den Unterschied zwischen abstraktem Konzept und konkretem Konstrukt der Sprache deutlich zu machen und damit allgemein auf eine kritische Einstellung gegenüber *jeder* Programmiersprache hinzuwirken.

"Konventionell" und "praxisnah" bedeuten freilich nicht, daß wir uns am Durchschnitt der heutigen Programmierpraxis orientieren. Im Gegenteil bemühen wir uns um begriffliche Klarheit und um eine angemessene Berücksichtigung der Theorie, beispielsweise durch die Behandlung der Themen Turing-Maschinen, Berechenbarkeit, Grammatiken, Semantik, Programmverifikation und Komplexität, auch wenn wir damit eine spätere Grundvorlesung Theoretische Informatik nur vorbereiten können.

Der Titel "Skriptum Informatik - eine konventionelle Einführung" soll auch den Werkzeug-Charakter betonen. Offensichtlich sind die Konflikte zwischen Systematik und Didaktik, zwischen erwünschter Vollständigkeit und notwendiger Beschränkung, nicht generell lösbar. Wir haben im Zuge der Vorbereitung immer wieder darüber gestritten, wie die Abschnitte zu ordnen seien, was wichtig und was entbehrlich ist. So liegt zwischen den Buchdeckeln nun ein Kompromiß, aus dem jeder Dozent, jeder Leser *seinen* Stoff in *seiner* Reihenfolge herausschälen möge.

In Vorbereitung befindet sich ein umfangreicher Übungsband mit zahlreichen Aufgaben und Musterlösungen, der ebenfalls bei vdF und Teubner erscheinen wird.

Lehrbuchautoren sind *Compiler* im ursprünglichen Wortsinn, sie tragen zusammen, übersetzen und verbinden, was andere geschaffen haben. Darum geht unser Dank an die Autoren aller Quellen, die wir - sicher nur in beschränktem Maße bewußt - verwendet haben. Ganz bewußt allerdings haben wir Quellen von Prof. Dr. V. Claus und Dr. A. Schwill (beide Oldenburg), insbesondere den Informatik-Duden (Bibliographisches Institut, Mannheim/Wien/Zürich, 1988), genutzt.

Viele Studierende und Mitarbeiter haben in den letzten Jahren durch Kritik und Korrekturen dabei geholfen, Rohfassungen in ein Buch zu verwandeln. Herr Dipl.-Inform. Rainer Götze hatte in der letzten Phase die anspruchsvolle Funktion des Redakteurs und Koordinators und damit wesentlichen Anteil am Zustandekommen. Frau Claudia Martsfeld hat mit Geduld und Einfühlungsvermögen große Teile des Manuskripts geschrieben. Ihnen allen gilt unser Dank.

Oldenburg/ Stuttgart, im August 1991

Hans-Jürgen Appelrath und Jochen Ludewig

Inhaltsverzeichnis

1. Grundlagen.....	9
1.1 Algorithmus und Berechenbarkeit	9
1.1.1 Algorithmus	9
1.1.2 Turing-Maschine.....	11
1.1.3 Berechenbarkeit.....	16
1.2 Sprache und Grammatik	19
1.2.1 Sprache.....	19
1.2.2 Grammatik.....	23
1.3 Rechner	29
1.3.1 Von-Neumann-Rechnerarchitektur	30
1.3.2 Rechnersysteme.....	36
1.4 Informatik als Wissenschaft.....	44
2. Imperative Programmierung - die Sprache MODULA-2.....	51
2.1 Syntaxdarstellungen	51
2.2 Elementare funktionale Modula-2-Programme	55
2.2.1 Eine MODULA-2-Teilsprache.....	55
2.2.2 Programmverzweigungen.....	66
2.2.3 Funktionen und Prozeduren	70
2.2.4 Elementare Datentypen, Aufzählungs- und Bereichstypen.....	72
2.2.5 Eingabevariablen.....	76
2.2.6 Rekursive Funktionen und Prozeduren	78
2.2.7 Nachteile funktional-rekursiver Programme.....	83
2.3 Iterative Programme	85
2.3.1 Wertzuweisungen und Referenz-Parameter.....	85
2.3.2 Gültigkeitsbereich und Lebensdauer.....	87
2.3.3 Anweisungen zur Iteration.....	94
2.3.4 Sprunganweisungen.....	100
2.3.5 Prozedurtypen.....	102
2.4 Komplexe Datentypen.....	107
2.4.1 Mengen (Sets).....	107
2.4.1.1 Darstellung und Manipulation von Mengen	108
2.4.1.2 Ein Beispiel für Sets	109
2.4.2 Arrays (Felder).....	111
2.4.3 Records (Verbunde)	117
2.4.3.1 Einfache Records.....	117
2.4.3.2 Records mit Varianten	123

2.4.4	Zeiger und dynamische Variablen.....	127
2.4.4.1	Die Speicherung auf der Halde	127
2.4.4.2	Operationen auf Zeigern	130
2.4.4.3	Verkettete Listen.....	133
2.4.4.4	Anwendungen und Probleme dynamischer Variablen ..	143
2.4.5	Dateien (Files).....	146
2.4.5.1	Eigenschaften und formale Beschreibung	146
2.4.5.2	Dateien in Pascal.....	148
2.4.5.3	Dateien in MODULA-2.....	152
3.	Abstraktion.....	155
3.1	Abstraktionskonzepte in Programmiersprachen.....	155
3.2	Abstraktion in MODULA-2.....	159
3.2.1	Das Prinzip der separaten Übersetzung	159
3.2.2	Modularisierung eines Programms	161
3.2.3	Datenkapselung	165
3.2.4	Abstrakte Datentypen.....	172
3.2.4.1	Das Prinzip des Abstrakten Datentyps	172
3.2.4.2	Abstrakte Datentypen Schlange und Stack	174
3.2.4.3	Abstrakter Datentyp für große Zahlen.....	181
3.2.4.4	Abstrakter Datentyp Komplex	186
4.	Semantik, Verifikation und Test	191
4.1	Konzepte für die Semantikdefinition	192
4.1.1	Semantik: Begriff und Motivation	192
4.1.2	Grundprinzipien der Semantiknotationen	193
4.1.3	Ein Beispiel für die operationale Semantik.....	196
4.2	Spezifikation und Verifikation von Programmen	202
4.2.1	Vor- und Nachbedingungen.....	202
4.2.2	Semantikdefinition einiger Anweisungen	205
4.2.3	Die Verifikation iterativer Programme.....	207
4.2.4	Beschreibung einer Schleife durch eine Invariante.....	213
4.2.5	Konstruktion iterativer Programme.....	215
4.2.6	Zusammenfassung.....	221
4.3	Test.....	222
4.3.1	Begriffsbildungen und Prinzipien.....	222
4.3.2	Grenzen des Tests	224
4.3.3	Die Konstruktion von Testdaten	226
4.3.4	Zusammenfassung.....	234

5. Programmierparadigmen und -sprachen	235
5.1 Programmierparadigmen.....	235
5.2.1 Imperatives Programmieren	237
5.2.2 Funktionales Programmieren.....	241
5.2.3 Logik-basiertes Programmieren.....	243
5.2.4 Objektorientiertes Programmieren	246
5.2.5 Regelbasiertes Programmieren	248
5.2 Übersicht über Programmiersprachen.....	252
6. Datenstrukturen und Algorithmen.....	255
6.1 Effizienz und Komplexität	255
6.1.1 Motivation und Begriffsbildung.....	255
6.1.2 Effizienz von Algorithmen.....	256
6.1.3 Komplexität von Funktionen und Sprachen	261
6.2 Graphen und Bäume.....	265
6.2.1 Graphen.....	265
6.2.2 Bäume.....	275
6.3 Suchen in gegebenen Datenstrukturen.....	287
6.3.1 Suchen in Tabellen.....	287
6.3.2 Suchen von Zeichenketten	291
6.4 Datenorganisationen für effizientes Suchen.....	302
6.4.1 Suchverfahren auf Bäumen.....	302
6.4.1.1 Binäre Suchbäume	302
6.4.1.2 AVL-Bäume.....	310
6.4.1.3 Optimale Suchbäume.....	326
6.4.1.4 B-Bäume	331
6.4.1.5 Weitere balancierte Suchbäume.....	340
6.4.2 Hashing	343
6.4.2.1 Begriffsbildungen und Anforderungen	343
6.4.2.2 Perfektes Hashing.....	345
6.4.2.3 Kollisionsbehandlung.....	346
6.4.2.4 Löschen in Hash-Tabellen.....	351
6.4.2.5 Aufwandsabschätzung.....	351
6.4.2.6 Implementierung von Kollisionsauflösungen.....	353
6.5 Sortieren	361
6.5.1 Klassifizierung und allgemeine Betrachtungen.....	361
6.5.2 Interne Sortiervverfahren.....	369
6.5.2.1 Einfache Sortiervverfahren	369

6.5.2.2	Schnelle Sortiervverfahren.....	373
6.5.2.3	Implementierung ausgewählter Sortiervverfahren	383
6.5.2.4	Aufwandsvergleich der Sortiervverfahren	388
6.5.2.5	Sortieren durch Streuen und Sammeln.....	389
6.5.3	Externe Sortiervverfahren	393
6.5.3.1	Direktes Mischen	394
6.5.3.2	Natürliches Mischen	395
6.5.3.3	Mehrwege-Mischen	395
6.6	Speicherverwaltung.....	397
6.6.1	Algorithmische Konzepte	398
6.6.2	Implementierung von Stacks	406
Anhang A: Mathematische Grundbegriffe und Formeln		413
Anhang B: Syntaxdiagramme für MODULA-2		417
Literatur		437
Abkürzungsverzeichnis		440
Index		441

1. Grundlagen

Durch dieses Kapitel sollen u.a. die folgenden Grundbegriffe der Informatik eingeführt werden:

- Algorithmus als Verarbeitungsvorschrift zur Lösung von Problemen;
- Turing-Maschine als universelles Maschinenmodell zur Realisierung von Algorithmen;
- Berechenbarkeit als Eigenschaft von Funktionen, sich durch Algorithmen realisieren zu lassen;
- Sprache als Ausdrucksmittel zur exakten Formulierung von Algorithmen;
- Grammatik als System zur Erzeugung von Sprachen;
- Rechner als Gerät, das in einer Sprache, speziell einer Programmiersprache codierte Algorithmen ausführt.

Dabei können wir manches nur anreißen, was in anderen Grundlagenvorlesungen, vor allem in "Theoretischer Informatik" und "Rechnerstrukturen" zu vertiefen sein wird. Der begriffliche Rahmen sollte aber in einer Informatik-Einführung abgesteckt werden, bevor das Thema "Programmierung" angesprochen wird.

1.1 Algorithmus und Berechenbarkeit

1.1.1 Algorithmus

Ein intuitiver Algorithmusbegriff

Unter einem *Algorithmus* versteht man eine präzise, endliche *Verarbeitungsvorschrift*, die so formuliert ist, daß die in der Vorschrift notierten *Elementaroperationen* von einer mechanisch oder elektronisch arbeitenden *Maschine* durchgeführt werden können. Die Anzahl der verfügbaren Elementaroperationen ist beschränkt, ebenso ihre Ausführungszeit. Aus der sprachlichen Darstellung des Algorithmus muß die Abfolge der einzelnen Verarbeitungsschritte eindeutig hervorgehen. Hierbei sind Wahlmöglichkeiten zugelassen. Nur muß dann genau festliegen, wie die Auswahl einer Möglichkeit erfolgen soll.

Beispiele für Algorithmen sind Vorschriften zum Addieren, Subtrahieren oder Multiplizieren von Zahlen sowie der euklidische Algorithmus zur Berechnung des größten gemeinsamen Teilers. Übliche Kochrezepte, Bastelanleitungen, Partituren, Spielregeln und alltägliche Vorschriften erinnern an Algorithmen; sie sind aber nicht exakt

ausformuliert und enthalten meist Teile, die unterschiedlich interpretiert werden können.

Ein Algorithmus beschreibt also eine Funktion (Abbildung) $f: E \rightarrow A$ von der Menge der zulässigen *Eingabedaten* E in die Menge der *Ausgabedaten* A . Vereinfacht kann man auch sagen, ein Algorithmus gibt an, wie Eingabedaten schrittweise in Ausgabedaten umgewandelt werden. Aber nicht jede Funktion läßt sich durch einen Algorithmus realisieren, wie wir im Abschnitt 1.1.3 ("Berechenbarkeit") sehen werden.

Typische Eigenschaften von Algorithmen

Algorithmen besitzen charakteristische Eigenschaften:

- (a) *Abstrahierung*
Ein Algorithmus löst i.a. eine Klasse von Problemen. Die Wahl eines einzelnen, aktuell zu lösenden Problems aus dieser Klasse erfolgt über Parameter.
- (b) *Finitheit*
Die Beschreibung eines Algorithmus selbst besitzt eine endliche Länge (statische Finitheit). Ferner darf zu jedem Zeitpunkt, zu dem man die Abarbeitung eines Algorithmus unterbricht, der Algorithmus nur endlich viel Platz belegen (dynamische Finitheit), d.h. die bei der Abarbeitung des Algorithmus entstehenden Datenstrukturen und Zwischenergebnisse sind endlich.
- (c) *Terminierung*
Algorithmen, die für jede Eingabe nach endlich vielen Schritten ein Resultat liefern und anhalten, heißen terminierend, sonst nicht-terminierend. In der Praxis interessieren häufig nur terminierende Algorithmen, aber in speziellen Bereichen wie z.B. bei Betriebssystemen (Basis-Algorithmen eines Rechners) oder bei der Überwachung von Anlagen oder Produktionsstätten sind auch nicht-terminierende Algorithmen von Bedeutung.
- (d) *Determinismus*
Ein Algorithmus heißt deterministisch, wenn zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit der Fortsetzung besteht. Hat ein Algorithmus an mindestens einer Stelle zwei oder mehr Möglichkeiten der Fortsetzung, von denen eine nach Belieben ausgewählt werden kann, so heißt er nichtdeterministisch. Kann man den Fortsetzungsmöglichkeiten Wahrscheinlichkeiten zuordnen, so spricht man von stochastischen Algorithmen.
- (e) *Determiniertheit*
Algorithmen sind i.a. determiniert, d.h. wird ein Algorithmus mit den gleichen Eingabewerten und Startbedingungen wiederholt, so liefert er stets das gleiche Ergebnis. Eine Erweiterung bilden die nicht-determinierten Algorithmen, die bei gleichen Startbedingungen unterschiedliche Ergebnisse berechnen können. Diese Eigenschaft nimmt man manchmal in Kauf, z.B. wenn ein exakter Lö-

sungsalgorithmus eine hohe Komplexität hat und man darum heuristische Methoden anwendet, also auf die erschöpfende Bearbeitung aller Fälle (und damit oft auf die absolut beste Lösung) verzichtet. Ein terminierender, deterministischer Algorithmus ist immer determiniert. Ein terminierender, nichtdeterministischer Algorithmus kann determiniert oder nicht-determiniert sein.

Eine faszinierende und zentrale Aufgabe der Informatik ist es, Probleme durch Algorithmen zu lösen. Ist ein solcher Lösungsalgorithmus entdeckt und formuliert, so ist das gestellte Problem aus Sicht der Theorie "erledigt", da jetzt scheinbar kein Problem mehr vorliegt. Wenngleich durch Algorithmen große Problembereiche als "gelöst", "entwertet" oder "nun uninteressant" erscheinen, so suchen Informatiker oft weiter nach einem *möglichst guten* Algorithmus, d.h. die *Effizienz* von Algorithmen spielt eine entscheidende Rolle:

Man kann beweisen, daß es zu jedem Algorithmus unendlich viele verschiedene, *äquivalente* Algorithmen gibt, die die gleiche Aufgabe lösen. Die Suche nach schnelleren oder kompakteren Algorithmen oder der Beweis, daß es solche nicht geben kann, führen meist auf komplizierte Probleme, für deren Lösung man keine generellen Methoden kennt.

Wichtiger als Laufzeit und Speicherbedarf sind oft andere, leider nicht ebenso einfach meßbare oder sichtbare Eigenschaften, beispielsweise die *Änderbarkeit*, die *Portabilität* (Übertragbarkeit eines Programms auf andere Rechner) und vor allem die *Zuverlässigkeit*. Da diese Qualitäten erst bei größeren, praxisrelevanten Programmen zum Tragen kommen, werden sie von den Studenten regelmäßig vernachlässigt.

Darstellung von Algorithmen

Algorithmen kann man darstellen, indem man sie in einer (wohldefinierten) *Programmiersprache* codiert, oder dadurch, daß man eine (exakt definierte) *Maschine* angibt, die den Algorithmus schrittweise nachvollzieht. Ein einfaches *Maschinenmodell* dafür ist z.B. die im nächsten Abschnitt vorgestellte Turing-Maschine.

Statt eine Maschine konkret zu bauen, genügt es auch, ihre Grundbefehle, ihren Speicher (für Daten) und ihre Kontrollstrukturen, d.h. die erlaubten Verknüpfungen der Grundbefehle, festzulegen. Man gibt also nur eine gedachte, eine sogenannte *virtuelle Maschine* an, von der man weiß, daß man sie *prinzipiell* auch bauen oder von einer realen Maschine *simulieren* lassen kann.

1.1.2 Turing-Maschine

Die Turing-Maschine (TM) ist ein *universelles*, d.h. nicht auf die Abarbeitung von Algorithmen einer bestimmten Klasse beschränktes Modell, das 1936 von dem Mathematiker A. M. Turing (1912 - 1954) vorgeschlagen wurde. Man beachte, daß wir heute in der TM unvermeidlich einen (sehr primitiven) Rechner sehen, während sie

von Turing als reines Gedankenmodell eingeführt worden war. Dazu trägt auch bei, daß das englische Wort "machine" (= Automat) mit "Maschine" übersetzt wurde. Manche Mißverständnisse (z.B. bezüglich des scheinbar bedrohlich großen Rechenaufwands der TM) haben hierin ihre Ursache.

Formale Definition einer TM

Die statischen Merkmale einer TM M können durch fünf Komponenten

$M = (B, I, Q, \alpha, \Delta)$ beschrieben werden:

- B ist das *Bandalphabet*, also eine endliche Menge unterschiedlicher Zeichen, mit denen das Band (siehe unten) beschrieben ist. Wir verwenden in den folgenden Beispielen kleine, schräggestellte Buchstaben, z.B. $B = \{a, b, c, z\}$. "z" hat als *Leerzeichen* spezielle Bedeutung (siehe unten).
- $I \subseteq B \setminus \{z\}$ ist das *Eingabealphabet*.
- Q ist eine endliche *Zustandsmenge*; die Zustände werden hier durch griechische Buchstaben bezeichnet, z.B. $Q = \{\alpha, \beta, \gamma\}$.
- $\alpha \in Q$ ist der *Anfangszustand*.
- $\Delta: Q \times B \rightarrow Q \times B \times \{L, R, H\}$ ist die *Übergangsfunktion*. L, R und H stehen für Links, Rechts und Halt.

Innerhalb einer speziellen, von einer TM durchgeführten Berechnung kommen drei Angaben hinzu:

- $q \in Q$ ist der *aktuelle Zustand*.
- $T: \mathbb{Z} \rightarrow B$ ist das *Band* („Tape“), formal eine Abbildung der Menge der ganzen Zahlen in B . Die ganzen Zahlen können hier als Numerierung der Zellen eines unendlich langen Bandes interpretiert werden, in jeder Zelle steht ein Zeichen. Bis auf einen endlich langen Abschnitt, der aber im Zuge der Berechnung wachsen oder schrumpfen kann, ist das Band mit Leerzeichen (z) besetzt. In diesem Abschnitt steht zu Beginn das *Eingabewort*, am Schluß das *Ergebniswort*.
- $C \in \mathbb{Z}$ ist die *Band-Position*, wo die TM gerade arbeitet. An dieser Stelle steht auf dem Band das Zeichen $t = T(C)$.

Ein *Arbeitsschritt* der Berechnung besteht nun darin, daß die Abbildung Δ auf (q, t) angewendet wird. Sie liefert den neuen Zustand q' und das neue Zeichen t' , das t ersetzt. Schließlich bedeuten L und R, daß C um 1 gesenkt bzw. erhöht wird (Schritt nach links bzw. rechts). Mit H bleibt die TM stehen (Halt). Eine vollständige Berechnung besteht darin, daß die TM im Zustand α an bestimmter Position C auf gegebenem Band T startet und Schritte ausführt, bis sie hält. Das *Resultat* der Berechnung ist dann die neue Belegung des Bandes T' , von der wieder nur der endlich lange Abschnitt interessiert, in dem andere Zeichen als das Leerzeichen z vorkommen.

Tritt im Zuge der Berechnung eine Kombination (q, t) auf, deren Abbildung Δ nicht definiert ist (Δ kann eine *partielle Funktion* sein), so bricht die Berechnung ab, und das Resultat ist undefiniert. Es ist ebenfalls undefiniert, wenn die TM nicht hält, sondern endlos weitere Schritte ausführt.

Ein anschauliches Modell der TM

Anschaulich kann man sich eine TM als ein sehr spezielles Tonbandgerät vorstellen, bei dem das nach beiden Seiten unbegrenzt lange Magnetband in einzelne Zellen unterteilt ist. In jeder Zelle des Bandes steht genau ein Zeichen. Es kann immer nur die eine Zelle gelesen und verändert werden, die sich gerade unter dem "Magnetkopf", genannt *Lese-Schreibkopf* (LSK) befindet. Das Band (oder der LSK) kann um eine Zelle nach rechts oder links *bewegt* werden. (Wir bevorzugen hier die anschaulichere Vorstellung, daß der LSK bewegt wird.)

Man beachte, daß nur der LSK ein endlicher Automat ist, nicht aber die ganze TM, denn auf dem unbegrenzten Band können beliebig viele verschiedene Situationen entstehen. Damit ist die primitiv erscheinende TM konzeptionell mächtiger als jeder reale, nur mit endlich großem Speicher ausgerüstete Rechner, und wir können - freilich ohne die Möglichkeit der praktischen Umsetzung - feststellen, daß irgend ein (sequentiell arbeitender) Computer nichts kann, was nicht auch eine geeignete TM könnte. Die Umkehrung dieses letzten Satzes ist es, was die TM so interessant macht: Was *sie* nicht kann, das kann auch kein Computer.

Der LSK der TM (oder in etwas ungenauer, aber üblicher Vereinfachung: die TM) ist zu jedem Zeitpunkt in einem bestimmten Zustand; die Menge der Zustände ist endlich und für eine bestimmte TM fest. Ein Arbeitsschritt der TM besteht darin, den Inhalt der Zelle, die gerade unter dem LSK liegt, zu lesen, ihn neu zu schreiben, den Zustand zu wechseln und einen Schritt nach links oder rechts zu tun oder zu halten. Das neue Zeichen, der neue Zustand und die Bewegung (oder das Halten) sind durch eine *Übergangstabelle* bestimmt. Diese enthält (höchstens) eine Zeile für jede Kombination aus Zustand und eingelesenem Zeichen, also bei einer TM mit s Zuständen und n Zeichen (höchstens) $s \cdot n$ Zeilen. Eine Übergangstabelle wird auch als *Turing-Programm* bezeichnet.

Abb. 1.1 Anschauliches Modell einer TM

Die Belegung des Bandes bedarf der Interpretation; bei der TM in Abb. 1.1 können wir beispielsweise festlegen, daß im *Eingabewort* *aaaabaa* (Zellen 18 - 24) die Sequenzen der Zeichen *a* natürliche Zahlen repräsentieren, hier also 4 und 2, getrennt durch ein *b*. Die Berechnung kann je nach Übergangstabelle eine Funktion davon liefern, etwa die Differenz. Dann wäre als Resultat *aa* zu erwarten.

Beispiel einer Berechnung

Für das genannte Beispiel der Subtraktion zweier natürlicher Zahlen (ohne die null) wäre eine geeignete Übergangstabelle das folgende Turing-Programm:

q	t	$\Delta(q, t)$		
α	<i>a</i>	β	<i>z</i>	R
β	<i>a</i>	β	<i>a</i>	R
β	<i>b</i>	γ	<i>b</i>	R
γ	<i>a</i>	δ	<i>b</i>	R
γ	<i>b</i>	γ	<i>b</i>	R
δ	<i>a</i>	ϵ	<i>a</i>	L
δ	<i>z</i>	ζ	<i>z</i>	L
ϵ	<i>a</i>	ϵ	<i>a</i>	L
ϵ	<i>b</i>	ϵ	<i>b</i>	L
ϵ	<i>z</i>	α	<i>z</i>	R
ζ	<i>a</i>	ζ	<i>a</i>	L
ζ	<i>b</i>	ζ	<i>z</i>	L
ζ	<i>z</i>	η	<i>z</i>	R
η	<i>a</i>	α	<i>a</i>	H

Die Zustände dieser TM lassen sich wie folgt anschaulich interpretieren:

α : den Minuenden dekrementieren

β : über den Minuenden nach rechts laufen bis zum ersten Zeichen *b*

γ : den Subtrahenden nach rechts suchen und dekrementieren

δ : feststellen, ob Subtrahend auf null reduziert ist, falls ja, Fortsetzung mit ζ , sonst mit ϵ

ϵ : zum nächsten Dekrementierungsschritt wieder nach links zum Anfang laufen

ζ : Zeichen rechts vom Resultat löschen, d.h. *b* durch *z* ersetzen, und an den Anfang laufen

η : am Anfang des Resultats halten.

Startet man die durch die vorstehende Tabelle gegebene TM in der Situation, wie sie in Abb. 1.1 gezeigt ist, also im Zustand α mit dem LSK über dem ersten Zeichen (von links) des Eingabewortes *aaaabaa*, so wird die erste Zeile der Tabelle angewandt, d.h. *a* wird durch *z* ersetzt, der LSK bekommt den Zustand β und geht nach rechts, und es entsteht die Situation, wie sie in Abb. 1.2 gezeigt ist.

Abb. 1.2 Situation der TM nach dem ersten Schritt

Nach weiteren vier Schritten, bei denen der LSK im Zustand β bleibt und das Band nicht verändert, liest er in Zelle 22 das *b* und wechselt in den Zustand γ (Abb. 1.3).

Abb. 1.3 Situation der TM nach dem fünften Schritt

Hier wird nun das *a* in Zelle 23 durch *b* ersetzt, der LSK geht nach rechts und in den Zustand δ . Da in Zelle 24 noch ein *a* steht, wird - nach dem Rücklauf im Zustand ϵ - der Vorgang wiederholt, so daß das *a* in Zelle 19 durch *z*, das in Zelle 24 durch *b* ersetzt wird. Dann beginnt die Ende-Behandlung (im Zustand ζ), die Zeichen *b* in den Zellen 24 bis 22 werden durch *z* ersetzt, und die Maschine hält schließlich auf Zelle 20 (Abb. 1.4).

Abb. 1.4 Situation der TM vor dem letzten Schritt (d.h. vor dem Halten)

Wir sehen an diesem Beispiel einige charakteristische Merkmale der TM:

- Die absolute Position auf dem Band hat keine Bedeutung, die Position des Eingabewortes kann also ohne jede Konsequenz versetzt werden, wenn der LSK

mitwandert. Das Ergebniswort weist dann die gleiche Verschiebung auf. Die Numerierung oben dient also nur der Beschreibung.

- Die Subtraktion natürlicher Zahlen ist eine ^{teilweise} partielle Funktion, d.h. sie ist nur definiert, wenn der Minuend größer ist als der Subtrahend; andernfalls ist das Ergebnis keine natürliche Zahl. Das Turing-Programm oben reflektiert dies durch fehlende Definitionen für Übergänge, die bei zulässigen Eingaben nicht benötigt werden. Startet man die TM aber mit dem Eingabewort *aba*, so wird beim achten Schritt $\Delta(\epsilon, z)$ abgefragt, und die Berechnung bricht mit undefiniertem Ergebnis ab. Eine andere Möglichkeit (unter vielen) wäre es, in die Tabelle die Zeile $\Delta(\epsilon, z) = (\epsilon, z, L)$ aufzunehmen, was zur Folge hätte, daß die TM nicht hält.
- Auch einfache Operationen erfordern viele Schritte. Sie können aber - bei entsprechenden Erweiterungen der Zustandsmenge und der Übergangstabelle - zu beliebig umfangreichen Algorithmen ausgebaut und kombiniert werden. Daß diese in der Praxis nicht simuliert werden können, weil sie sehr viel Speicher und sehr viele Schritte benötigen, tut der Theorie keinen Abbruch.
- Wählt man Bandalphabet und Turing-Programm anders, so sind beliebige Zahlendarstellungen möglich, z.B. auch die dezimale. Die Theorie zeigt aber, daß auch eine TM mit nur zwei Zeichen ($B = \{a, z\}$) bei entsprechend mehr Zuständen grundsätzlich die gleiche Mächtigkeit hat wie jede andere. Im Beispiel kommt man mit veränderter Übergangstabelle leicht ohne das Zeichen *b* aus.
- Durch entsprechende Codierungen können beliebige, aber exakt beschriebene Problemstellungen mit TMn bearbeitet werden. Im Beispiel waren natürliche Zahlen codiert worden, man kann aber ebenso auch Wahrheitswerte, Texte oder andere Informationen darstellen. Für die Theorie ist vor allem die Möglichkeit wichtig, eine TM selbst, d.h. ihre Tabelle, als Eingabewort zu bearbeiten. Damit lassen sich TMn konstruieren, die andere TMn analysieren (vgl. 1.1.3, die nicht-berechenbaren Funktionen).

1.1.3 Berechenbarkeit

Berechenbar heißt eine Funktion $f : M \rightarrow N$ dann, wenn es einen Algorithmus gibt, der für jeden Eingabewert $m \in M$, für den $f(m)$ definiert ist, nach endlich vielen Schritten anhält und als Ergebnis $f(m)$ liefert. In allen Fällen, in denen $f(m)$ nicht definiert ist, bricht der Algorithmus nicht ab.

Eine berechenbare Funktion $f : M \rightarrow N$, die total, d.h. für jedes $m \in M$ definiert ist, heißt auch *rekursive Funktion* (bezüglich M). Berechenbare Funktionen f , die nicht für alle Elemente $m \in M$ definiert sind, nennt man auch *partiell-rekursive Funktionen*.

Es liegt nahe zu vermuten, daß jede Funktion, die man präzise definieren kann, auch berechenbar ist. Überraschenderweise ist dies nicht der Fall.

Berechenbare Funktionen

Oben war die Subtraktion gezeigt worden. Offenbar handelt es sich um eine partiell-rekursive Funktion. Die Addition wäre dagegen eine rekursive Funktion, denn die Summe natürlicher Zahlen liefert stets eine natürliche Zahl. Weitere bekannte Beispiele berechenbarer Funktionen sind

- (a) Bestimmung des größten gemeinsamen Teilers (ggT) zweier natürlicher Zahlen mit dem euklidischen Algorithmus:
 $\text{ggT} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mit
 $\text{ggT}(a, b) = \text{„größter gemeinsamer Teiler von } a \text{ und } b\text{“}$
- (b) Feststellung, ob eine natürliche Zahl Primzahl ist:
 $\text{prim} : \mathbb{N} \rightarrow \{\text{TRUE}, \text{FALSE}\}$ mit
 $\text{prim}(n) = \text{TRUE}$, falls n Primzahl ist, sonst FALSE
- (c) Sortieren von r natürlichen Zahlen:
 $\text{sort} : \mathbb{N}^r \rightarrow \mathbb{N}^r$ mit $\text{sort}(x_1, \dots, x_r) = (x_{i_1}, x_{i_2}, \dots, x_{i_r})$, wobei
 $i_1, \dots, i_r \in \{1, \dots, r\}$ paarweise verschieden sind und $x_{i_1} \leq x_{i_2} \leq \dots \leq x_{i_r}$ gilt.

Nicht-berechenbare Funktionen

Nicht-berechenbare Funktionen sind nicht ganz so einfach, aber durchaus anschaulich. Daß sie nicht nur *schwer* zu realisieren sind, sondern prinzipiell gar nicht, beweist die Theorie der Turing-Maschinen. Wir können den Beweis hier nicht wiedergeben; sein Prinzip besteht darin zu zeigen, daß es zu einem Widerspruch führt, wenn man die Existenz von Lösungen der folgenden und ähnlicher Probleme annimmt:

Sei A die Menge aller Algorithmen und E die Menge aller Eingabewörter. Die - in geeigneter Form codierte - Tabelle der Subtraktionsmaschine gehört also zu A , die Wörter *aaaabaa* und *aba* gehören zu E .

(a) Halteproblem:

$f : A \times E \rightarrow \{\text{TRUE}, \text{FALSE}\}$ mit

$f(a, e) = \text{TRUE}$, falls der Algorithmus a , angesetzt auf die Eingabe e , nach endlich vielen Schritten hält, sonst FALSE.

Es gibt keinen Algorithmus, der f berechnet, also als Eingabe einen beliebigen anderen Algorithmus und dessen Daten erhält und feststellt, ob die Berechnung terminieren wird.

Man beachte, daß es natürlich möglich ist, dies für *sehr viele* Algorithmen zu entscheiden, insbesondere - durch Probieren - für alle terminierenden. Führt das

Probieren aber nach einer gewissen Zahl von Schritten nicht zum Halt, so beweist das nichts, denn man kann auch keine Obergrenze angeben, bis zu der der Algorithmus höchstens noch terminieren kann.

(b) **Äquivalenzproblem:**

$g : A \times A \rightarrow \{\text{TRUE}, \text{FALSE}\}$ mit

$g(a_1, a_2) = \text{TRUE}$, falls die Algorithmen a_1 und a_2 die gleiche Funktion berechnen (also für beliebige Eingaben das gleiche oder kein Resultat berechnen), sonst FALSE.

Es gibt keinen Algorithmus für g , der als Eingabe zwei andere Algorithmen erhält und entscheidet, ob die beiden Algorithmen dasselbe leisten, d.h. dieselbe Funktion realisieren.

Bedeutung der TM als Modell, Churchsche These

Abschließend wollen wir den wichtigen Zusammenhang zwischen Turing-Maschinen und berechenbaren Funktionen herstellen.

Das Besondere an dem einfachen Modell einer TM ist, daß es sich als äquivalent zu allen bekannten Formalisierungen des Begriffs "Berechenbarkeit" erwiesen hat, d.h. alle bekannten Formalisierungen des Begriffs "Algorithmus" lassen sich in das Modell der TM überführen und umgekehrt.

Dies veranlaßte Turing und Church bereits 1936 zu der Vermutung, daß es zu jedem berechenbaren Algorithmus (der eine Funktion f realisiert) eine TM M gibt, so daß für die durch M repräsentierte Funktion f' gilt: $f = f'$. Mit anderen Worten: jeder Algorithmus läßt sich durch ein Turing-Programm realisieren. Oder auch: jede berechenbare Funktion läßt sich programmieren. Diese Behauptung läßt sich mathematisch nicht beweisen, weil die Begriffe "Algorithmus" und "berechenbare Funktion" mathematisch nicht faßbar sind. Sie wurde darum "nur" als These, als *Churchsche These*, bekannt. Diese These konnte bisher nicht widerlegt werden und ist allgemein anerkannt. Die TM wurde damit zur Grundlage der Theorie der Berechenbarkeit.

Im Rahmen der Informatik-Einführung genügt diese Erkenntnis: Wer die Arbeitsweise einer TM verstanden hat, kennt prinzipiell die Klasse der Algorithmen, also die Menge aller berechenbaren, in einer Programmiersprache codierbaren Funktionen, kennt ihre Grenzen und weiß, daß es auch nicht-berechenbare Funktionen gibt.

Daß man in der Praxis nur Interesse an einer mikroskopisch kleinen Menge berechenbarer Funktionen hat (neben einem ausreichenden "Anwendungsbezug" spielen insbesondere Komplexitätsfragen wie "Ist die Funktion in akzeptabler Zeit berechenbar?" eine Rolle), steht auf einem ganz anderen Blatt ...

1.2 Sprache und Grammatik

Abschnitt 1.1 sollte die zentrale Bedeutung des Algorithmusbegriffs in der Informatik deutlich gemacht haben. Er hat auch gezeigt, daß ein Algorithmus, beispielsweise als Übergangsfunktion formuliert, zur Ausführung noch einer Maschine bedarf, in diesem Falle der Turing-Maschine. Wir benötigen allgemeiner also eine sprachliche Formulierung des Algorithmus und eine Maschine (einen Automaten), der diese Formulierung interpretieren, "ausführen" kann. Zur Maschine gehört auch ein Speicher, bei der TM ist dies das Band.

Als Maschinen interessieren in der Informatik - zumindest in der praktischen, ingenieurwissenschaftlichen - reale Rechner, also nicht die Turing-Maschinen. Rechner - deren grobe Architektur in Abschnitt 1.3 vorgestellt wird - realisieren Algorithmen durch schrittweise Abarbeitung von Programmen, die in einer *Programmiersprache* codiert sind. Wer also Rechner programmieren will, muß *Syntax* und *Semantik* von Programmiersprachen kennen, um Algorithmen syntaktisch korrekt und im Sinne der beabsichtigten Berechnung zu codieren.

Im diesem Abschnitt wird der syntaktische Teil von Sprachen durch Einführung des Grammatikbegriffs präzisiert. Auf dieser Grundlage werden wir dann ab Kapitel 2 schon mit der Einführung in die "Programmierung" als Fertigkeit, Algorithmen in einer Programmiersprache zu formulieren und auf einem Rechner zu implementieren, beginnen. Formale Methoden zur Definition der Semantik von Programmiersprachen werden erst in Kapitel 4 vorgestellt.

1.2.1 Sprache

Bevor wir uns den verschiedenen Aspekten des Sprachbegriffs zuwenden, ist eine Vorbemerkung erforderlich: Der Begriff ist schillernd und wird in unterschiedlichen Bedeutungen gebraucht, auch in diesem Buch. Denn zunächst war und ist eine Sprache, wie das Wort sagt, das *Gesprochene*. In der Informatik spielt aber gerade das gesprochene Wort bis heute keine Rolle, so daß "Sprache" hier jede Konvention zur Kommunikation bezeichnet. Damit haben auch Verkehrsampeln, Heulbojen und Autos (Pedalanordnung) ihre Sprachen. In der Informatik stehen natürlich solche Sprachen im Vordergrund, die für die Kommunikation zwischen Menschen und Rechnern (in allen vier Kombinationen) taugen. Diese haben traditionell Schrift-Form, in den letzten Jahren zunehmend auch graphische Form.

Die Syntax einer Sprache bestimmt, welche Aussagen darin zulässig sind, die Semantik, welche Bedeutung die zulässigen Aussagen (und nur diese) haben. In der Praxis heißt das: Um ein Programm in die Notation des Rechners übertragen zu können, muß es zunächst syntaktisch in Ordnung sein; dies prüft der Compiler. Das er-

zeugte Maschinen-Programm repräsentiert die Semantik des Programms; bei syntaktischen Fehlern wird kein Maschinen-Code erzeugt.

Man kann sich diese Begriffsabgrenzung auch an folgender Analogie klarmachen: Durch Presse und Plakate wird bekanntgemacht, welche Eintrittskarten an einem bestimmten Tage im Opernhaus gültig sind. Wer keine gültige Karte vorweisen kann, wird nicht eingelassen (Syntax-Fehler), wer es kann, hat auch Anspruch auf einen bestimmten Platz (das ist die Semantik der Eintrittskarte).

Die Oper, die er dann geboten bekommt, also die Bedeutung der Eintrittskarte über ihren formalen Sinn hinaus (Zuordnung eines Platzes), könnte man als die *Pragmatik* bezeichnen. Auch beim Programmieren liegt der Sinn ja letztlich nicht in einer Zeichentransformation, sondern in Auswirkungen auf die Welt, beispielsweise einer Zahlung, dem geordneten Verkehrsfluß auf einer Kreuzung oder dem ästhetisch befriedigenden Satz eines Textes. Die Informatik tut sich allerdings mit der Pragmatik schwer, denn sie liegt ja außerhalb ihrer "Mauern". Auch wir werden uns im Rahmen dieses Buches mit der Semantik begnügen.

Eine formale Sprache ist nun, wie unten gezeigt wird, viel weniger als eine Sprache im hier beschriebenen Sinne, nämlich kaum mehr als die Syntax. Zu Programmiersprachen gehört dagegen auch immer die Semantik; allerdings ist die Abgrenzung zwischen Syntax und Semantik oft verwischt, weil man nur das zur Syntax rechnet, was sich in einen bestimmten Definitionsmechanismus fassen läßt (nämlich den der kontextfreien Sprachen). In diesem Falle prüft der Compiler mehr, als die Syntax verlangt.

Ein Beispiel hierfür ist die Syntax des Tagesdatums. Eine "3" als Zehnerziffer der Tagesnummer ist nur unter bestimmten Bedingungen zulässig, sie darf nur von "0" oder "1" gefolgt sein. Will man nun die Syntax der Tagesnummer möglichst einfach definieren, so beschränkt man sich darauf, als erstes Zeichen eine Ziffer bis "3" oder ein Leerzeichen zuzulassen, als zweite eine beliebige Ziffer. Dann muß man aber durch zusätzliche Bedingungen Nummern wie 0 oder 32 verbieten.

Künstliche Sprachen

Künstliche Sprachen - im Gegensatz zu natürlichen Sprachen wie Deutsch oder Englisch - wurden Ende des 19. Jahrhunderts entwickelt, um Fakten, Denkabläufe und Schlußfolgerungen formal beschreiben und analysieren zu können. Neben diesen Logik-Sprachen (*Aussagenlogik*, *Prädikatenlogik*) entstanden mit der Entwicklung von Rechnern seit 1940 Programmiersprachen, die ebenfalls präzise definiert werden mußten.

Künstliche Sprachen werden nach Regeln aufgebaut (Syntax, Grammatik), und ihre Wörter und Sätze besitzen eine wohldefinierte Bedeutung (Semantik). Während sich bei natürlichen Sprachen die Wörter, die Regeln und die Bedeutungen im Laufe der

Jahre ändern, besitzen künstliche Sprachen ein festes endliches Grundvokabular und eine feste Syntax und Semantik.

Beispiele

- (a) Aus dem Bereich der *natürlichen Sprachen*, denen eine klare Definition der Grammatik fehlt: Anfang und Schluß eines Dada-Gedichts (Kurt Schwitters: Der Bahnhof (1918)):

Man hat eine Leiter zur Sonne gestellt	(1)
Die Sonne ist schwarz	(2)
Die Mühle blüht	(3)
...	
Du deiner dir dich	(11)
Du deiner dich dir.	(12)

Die beiden ersten Zeilen sind syntaktisch korrekt (nur der Schlußpunkt fehlt jeweils), die Semantik ist klar, der Inhalt ist allerdings zweifelhaft. Zeile 3 ist ebenfalls syntaktisch korrekt, die Semantik ist aber unklar. (Solche Fälle sucht man bei Programmiersprachen zu vermeiden, nicht immer mit Erfolg!) Die beiden letzten Zeilen sind syntaktisch nicht korrekt, ihre Semantik bleibt damit undefiniert.

- (b) Die natürlichen Zahlen (ohne Null), dargestellt im Dezimalsystem mit arabischen (eigentlich: indischen) Ziffern, bilden eine einfache *künstliche Sprache* mit folgender Festlegung:

Syntax: Jede Zahl ist eine Sequenz von Ziffern (0, 1, ..., 9), wobei die erste Ziffer nicht 0 ist.

Semantik: Der Wert einer Zahl ist definiert als der Wert ihrer letzten Ziffer, vermehrt um den zehnfachen Wert der links davon stehenden Zahl, falls diese vorhanden ist.

Die von Niklaus Wirth von der ETH Zürich Anfang der 80er Jahre eingeführte Programmiersprache MODULA-2 ist eine künstliche Sprache, denn der Zeichenvorrat ist endlich, und es gibt eine *endliche Regelmenge*, die festlegt, ob ein Programmtext ein zulässiges MODULA-2-Programm ist oder nicht. Im Gegensatz zu natürlichen Sprachen rechnet man bei künstlichen Sprachen auch das Grundvokabular zum *Alphabet* und spricht statt von den Sätzen von *Wörtern* der Sprache, im Falle von Programmiersprachen von *Programmen*, d.h. ein syntaktisch korrektes Programm ist ein Wort einer künstlichen Sprache.

Zeichen, Alphabete, Wörter, freies Monoid und Sprache

Ein *Zeichenvorrat* ist eine endliche Menge unterscheidbarer, in bestimmtem Kontext nicht weiter zerlegbarer Objekte (z.B. Buchstaben, Bilder, Symbole, Töne). Elemente eines Zeichenvorrats heißen *Zeichen*.

Ein *Wort* ist eine endliche Folge von Zeichen. Seien A ein Zeichenvorrat und $\leq$ eine lineare Ordnung auf A . Dann heißt die linear geordnete Menge $(A, \leq)$ ein *Alphabet*.

Man beachte, daß das deutsche ABC kein Alphabet im Sinne der Bezeichnung ist, denn für einige Zeichen (z.B. Umlaute, "ß") fehlt teilweise die klare Regelung der Einordnung (vgl. die verschiedenen Formen von "Müller" im Telefonbuch). In Deutschland gibt es hierfür eine eigene Norm (DIN 5007).

Eine *Ziffer* ist ein Zeichen, das eine Zahl repräsentiert. *Zahlen* lassen sich auf verschiedene Art darstellen. Wir verwenden heute ganz vorrangig das Prinzip des *Dezimalsystems*, allgemeiner ein *Positionssystem*. Es hat folgende Kennzeichen: eine einheitliche Basis g aus der Menge der natürlichen Zahlen, unterschiedliche Ziffern für jede Zahl von 1 bis $g-1$ und eine spezielle Ziffer für null.

Die Ziffern haben ein von ihrer Position abhängiges Gewicht (Wert): Die letzte (rechte) Ziffer hat das Gewicht 1, alle anderen Ziffern haben das g -fache Gewicht der rechts folgenden Ziffer. Eine Zahl k ($g^{n-1} \leq k < g^n$) ist darstellbar durch n Ziffern. Mit $g=10$ haben wir das Dezimalsystem, mit $g=2$ das *Dualsystem*.

Die Menge aller Wörter, die sich durch die Hintereinanderreihung endlich vieler, nicht notwendig verschiedener Zeichen aus einem vorgegebenen Alphabet A ergeben, wird mit A^* (dem sogenannten *freien Monoid* über A) bezeichnet. Hierin ist auch das leere Wort ϵ enthalten, das aus keinem einzigen Alphabetzeichen besteht (ϵ ist nur ein Symbol für "nichts"). Man bezeichnet nun jede Teilmenge $L \subseteq A^*$ als *Sprache* über A .

Beispiel: Es ist $\{0, 1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$.

$L = \{0, 1, 10, 11, 100, 101, \dots\} \subset \{0, 1\}^*$, die Menge der Binärdarstellungen natürlicher Zahlen (mit der Null und ohne führende Nullen), ist eine Sprache über dem Alphabet $\{0, 1\}$.

Formale Sprachen

Die Syntax von künstlichen Sprachen wird durch sogenannte formale Sprachen über einem Alphabet beschrieben. Unter *formalen Sprachen* versteht man in der Informatik eine Sprache L zusammen mit einer Definitionsvorschrift, die *konstruktiv* ist. "Konstruktiv" bedeutet, daß die Definitionsvorschrift die Form einer sogenannten *Grammatik* hat, die L *erzeugt*, oder die Form einer *Maschine*, die L *erkennt*. Im folgenden wird zunächst ausführlich der in der Praxis wichtigere Ansatz des Spracherzeugens durch Grammatiken, später noch kurz der korrespondierende Ansatz des Spracherkennens durch Maschinen vorgestellt.

1.2.2 Grammatik

Zur Festlegung der Syntax einer Sprache verwendet man in aller Regel eine Grammatik. Diese besteht - vereinfachend gesagt - aus *Regeln*, die bestimmen, welche Wörter zur Sprache gehören und welche nicht. Für natürliche Sprachen (z.B. Deutsch, Englisch) ist eine präzise Festlegung der Syntax unmöglich, da die Sprachen nicht nur sehr umfangreich und kompliziert, sondern auch ständig im Fluß und individuell verschieden sind. Es entwickeln sich laufend neue Wörter (z.B. "Null-Wachstum"), die Schreibweise von Wörtern ändert sich (z.B. früher: "Blouse", heute "Bluse"), und mancher hält "Software" für ein deutsches Wort, ein anderer "aussitzen".

Da zur Verständigung mit Computern stets eine hinreichend präzise Definition der Syntax einer Sprache erforderlich ist, eignen sich natürliche Sprachen hierzu nur sehr beschränkt. Man hat daher einfachere Sprachen für Computer entwickelt, die sogenannten Programmiersprachen. Ihre Syntax kann präzise durch Grammatiken definiert werden.

Definition einer Grammatik

Eine *Grammatik* G für eine Sprache definiert man durch Angabe eines *Viertupels* (V, T, P, S) mit

- (1) T : Menge von Terminalsymbolen.

Terminalsymbole oder Terminalzeichen sind Zeichen oder Zeichenfolgen des Alphabets, aus dem die Wörter der Sprache bestehen. Jedes Wort der Sprache ist also eine Folge von Terminalsymbolen.

Beispiele: Terminalsymbole der deutschen Sprache in diesem Sinne sind alle deutschen Wörter einschließlich aller Modifikationen, die z.B. durch Deklination und Konjugation entstehen, also z.B. Auto, der, die, das, Haus, laufen, schreiben, Duden, Textes, waren.

Terminalsymbole in der Programmiersprache MODULA-2 sind u.a. BEGIN, END, :=, ,, INTEGER.

- (2) V : Menge von Nichtterminalsymbolen.

Nichtterminalsymbole oder *syntaktische Variablen* sind Zeichen oder Zeichenfolgen, die Abstraktionen der Programmkomponenten darstellen, also syntaktische Begriffe wie "Hauptsatz". Diese Wörter kommen selbst natürlich nicht in das Programm, sondern werden durch Anwendung der Produktionsregeln (siehe unten) schrittweise ersetzt, bis nur noch Terminalsymbole übrig sind.

Beispiele: Nichtterminalsymbole der deutschen Sprache sind u.a. Satz, Nebensatz, Subjekt, Prädikat, Objekt, Artikel.

Nichtterminalsymbole in der Programmiersprache MODULA-2 sind z.B. Number, Block, StatementSequence, ProgramModule (Anmerkung: wir verwenden für die Nichtterminalsymbole durchgängig die englischen Bezeichnungen).

(3) P: Menge von Produktionsregeln.

Eine *Produktionsregel* (oder kurz Regel) legt fest, wie man aus bereits bekannten Konstrukten oder Sätzen neue Konstrukte oder Sätze erhält.

Beispiele: Eine Regel der deutschen Sprache ist etwa "Verbindet man zwei Sätze S und T durch die Zeichenfolge ", und", so erhält man wieder einen Satz". Z.B. ist für S = "das Auto ist rot" und T = "es regnet" dann auch "S, und T", also "das Auto ist rot, und es regnet" ein Satz.

Eine Regel in der Programmiersprache MODULA-2 lautet: "Wenn S eine Anweisungsfolge und B eine Bedingung ist, so ist auch die Zeichenfolge "REPEAT S UNTIL B" eine Anweisung."

Regeln schreibt man i.a. nicht umgangssprachlich, sondern formalisiert auf, z.B.

- $(X_1 X_2 \dots X_n, Y_1 Y_2 \dots Y_m)$, also als *Paar*, dessen linker Teil die zu ersetzende Zeichenfolge und dessen rechter Teil die resultierende Zeichenfolge ist oder
- $X_1 X_2 \dots X_n \rightarrow Y_1 Y_2 \dots Y_m$, also als *Folgerung*, wobei $X_1, \dots, X_n, Y_1, \dots, Y_m$ Nichtterminalsymbole oder Terminalsymbole sind. Da Nichtterminalsymbole und Terminalsymbole wie auch die speziellen Symbole der Produktionsregeln (die sogenannten "*Metasymbole*") häufig aus den gleichen Zeichen (Buchstaben, Klammern usw.) bestehen, muß man durch spezielle Maßnahmen Unklarheiten ausschließen. Beispielsweise kann man verschiedene Schrifttypen verwenden, die Nichtterminalsymbole klammern (z.B. «Satz» für das Nichtterminalsymbol Satz) oder - wie in den folgenden Beispielen - die Terminalsymbole durch Klammerung unterscheiden (z.B. "x" für das Terminalsymbol x).

Beispiele: (a) Die obige Regel für die deutsche Sprache notiert man so:
(Satz, Satz ", und" Satz) oder so: Satz $\rightarrow$ Satz ", und" Satz
(b) Die Regel für MODULA-2 schreibt man entsprechend als:
(Anweisung, "REPEAT" Anweisung "UNTIL" Bedingung)
oder als
Anweisung $\rightarrow$ "REPEAT" Anweisung "UNTIL" Bedingung

- (4) S: Ein Startsymbol oder eine Menge von Startsymbolen.
 Das *Startsymbol* ist ein Nichtterminalsymbol, gewissermaßen das allgemeinste Konstrukt der Sprache, aus dem alle Wörter der Sprache erzeugt werden; in vielen Sprachen (z.B. in der Programmiersprache Pascal) ist dieses Startsymbol «Program». In MODULA-2 gibt es drei Startsymbole, weil es drei verschiedene Arten von Übersetzungseinheiten gibt («ProgramModule», «DefinitionModule», «ImplementationModule»).

Chomsky-Grammatiken

Grammatiken wurden in der oben charakterisierten Form im Jahre 1959 vom amerikanischen Linguisten Noam Chomsky (ausgesprochen: "Tschomsky") eingeführt. Man spricht daher auch häufig von Chomsky-Grammatiken.

Eine (*Chomsky-*) *Grammatik* G definiert man formal als Viertupel $G = (V, T, P, S)$, für das gilt:

- (1) V ist eine endliche, nichtleere Menge von Zeichen, den sogenannten *Nichtterminalsymbolen*.
- (2) T ist eine endliche, nichtleere Menge von Zeichen, den sogenannten *Terminalsymbolen*.
 V und T sind disjunkt, d.h. $V \cap T = \emptyset$.
- (3) S ist ein Element von V und heißt *Startsymbol*.
- (4) P ist eine endliche Menge von Paaren der Form (p, q) , wobei gilt:
 p und q sind Wörter aus Nichtterminalsymbolen und Terminalsymbolen, d.h. $p, q \in (V \cup T)^*$, wobei p mindestens ein Nichtterminalsymbol enthalten muß.
 P wird *Produktionssystem* oder *Regelsystem*, die Elemente von P werden *Produktionsregeln*, *Produktionen*, *i.Ersetzungsregeln* oder *Regeln* genannt.

Bemerkungen

1. Grammatiken ordnet man, je nachdem, welche Form die Produktionsregeln haben, in verschiedene Klassen ein. Man unterscheidet z.B. kontextsensitive, kontextfreie und reguläre Grammatiken.
 Die wichtigste Klasse bilden die *kontextfreien Grammatiken*, die für die Definition der Syntax von Programmiersprachen überwiegend verwendet werden. Bei kontextfreien Grammatiken muß auf der linken Seite jeder Produktionsregel genau ein Nichtterminalsymbol stehen.
2. Zur Darstellung von kontextfreien Grammatiken hat man eine Reihe von Formalismen entwickelt (die obige Paar- bzw. Pfeilschreibweise sind Beispiele), von denen *Syntaxdiagramme* und die *Backus-Naur-Form* die wichtigsten sind (wir werden sie zu Beginn von Kapitel 2 kennenlernen).

Beispiele

- (a) Die kontextfreie Grammatik $G_1 = (N_1, T_1, P_1, S_1)$ definiert einen kleinen Auszug der deutschen Sprache. Sei

$N_1 = \{\text{Satz, Subjekt, Prädikat, Objekt, Artikel, Substantiv, Blank}\},$

$T_1 = \{\text{"die ", "Maus ", "Katze ", "jagt ", "beißt ", " "}\},$

$P_1 = \{(\text{Satz} \rightarrow \text{Subjekt Prädikat Objekt "."}),$

$(\text{Subjekt} \rightarrow \text{Artikel Substantiv}),$

$(\text{Prädikat} \rightarrow \text{"jagt "}),$

$(\text{Prädikat} \rightarrow \text{"beißt "}),$

$(\text{Artikel} \rightarrow \text{"die "}),$

$(\text{Objekt} \rightarrow \text{Artikel Substantiv}),$

$(\text{Substantiv} \rightarrow \text{"Maus "}),$

$(\text{Substantiv} \rightarrow \text{"Katze "})\},$

$S_1 = \text{Satz}.$

In den Produktionen sind wie in den Beispielen oben die Terminalsymbole in Häkchen geklammert. Die erste Regel besagt beispielsweise, daß das grammatische Konstrukt Satz aus einem Subjekt, einem Prädikat und einem Objekt (in dieser Reihenfolge) besteht und mit einem Punkt enden muß. Die beiden Regeln für «Substantiv» definieren die Wörter «Katze» und «Maus» als Substantive.

- (b) $G_2 = (N_2, T_2, P_2, S_2)$ mit

$N_2 = \{A, B\}, T_2 = \{a, b, c, d\},$

$P_2 = \{(A \rightarrow a B bc), (B \rightarrow a B b), (a B b \rightarrow d)\},$

$S_2 = A.$

G_2 ist keine kontextfreie Grammatik, da die dritte Produktionsregel auf der linken Seite mehr als nur das Nichtterminalsymbol enthält. In diesem speziellen Fall kann man daraus aber eine kontextfreie, die gleiche Sprache erzeugende Grammatik machen, indem man P_2 ersetzt durch P_2' mit

$P_2' = \{(A \rightarrow B c), (B \rightarrow a B b), (B \rightarrow d)\}.$

Ableitungsbegriff; die von einer Grammatik erzeugte Sprache

Als nächstes präzisieren wir, wie man mit einer Grammatik die syntaktisch korrekten Wörter gewinnt.

Sei G eine Grammatik. Dann ist ein *Ableitungsschritt* von G wie folgt definiert:

Sei x ein Wort, das aus Nichtterminalsymbolen und Terminalsymbolen von G gebildet ist, d.h. $x \in (V \cup T)^*$, und sei (p, q) eine Regel aus der Menge der Produktionsregeln von G , so daß p als *Teilwort* von x auftritt. x ist also darstellbar als $x = x'px''$ mit $x', x'' \in (V \cup T)^*$. Dann ist das Ergebnis der Anwendung von (p, q) auf x das Wort y , das entsteht, wenn man aus x das Teilwort p aus seinem Kontext

$x' \dots x''$ herausnimmt und an seine Stelle das Wort q aus der Regel einsetzt, d.h. $y = x'qx''$. Da p in x mehrfach, sogar überlappend vorkommen kann, sind verschiedene Anwendungen möglich.

Man schreibt dann $x \rightarrow_{(p,q)} y$ bzw. nur $x \rightarrow y$, wenn die konkrete Regel (p, q) nicht interessiert, und sagt: " x ist *ableitbar* nach y *durch Anwendung der Regel* (p, q) ". Auf ein Wort x_0 können mehrere Ableitungsschritte nacheinander angewendet werden, d.h., daß es eine Folge von Wörtern $x_1, x_2, \dots, x_n$ geben kann, so daß gilt: $x_0 \rightarrow x_1, x_1 \rightarrow x_2, \dots, x_{n-1} \rightarrow x_n$ ($n \geq 0$). Man sagt in diesem Fall: x_0 ist *ableitbar* nach x_n *bezüglich* G und schreibt dies z.B. als $x_0 \xrightarrow{*}_G x_n$.

Die Folge $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_{n-1} \rightarrow x_n$ heißt *Ableitung (der Länge n)* von x_0 nach x_n bezüglich G .

Hiermit läßt sich nun jeder Grammatik $G = (V, T, P, S)$ eine (formale) Sprache zuordnen: Die von G *erzeugte Sprache* (Bezeichnung: $L(G)$) ist die Menge aller Wörter, die aus dem Startsymbol ableitbar sind und nur aus Terminalsymbolen bestehen.

Also: $L(G) := \{w \mid S \xrightarrow{*}_G w \wedge w \in T^*\}$.

Beispiel

Sei G_1 die Grammatik aus obigem Beispiel. Das Wort (der Satz) "die Katze jagt die Maus." ist in G_1 aus dem Startsymbol «Satz» ableitbar. Eine Ableitung hierfür ist z.B. (jeder Pfeil symbolisiert den Ableitungsschritt bezüglich einer Regel)

Satz $\rightarrow$ Subjekt Prädikat Objekt "."
 $\rightarrow$ Artikel Substantiv Prädikat Objekt "."
 $\rightarrow$ Artikel "Katze" Prädikat Objekt "."
 $\rightarrow$ Artikel "Katze jagt" Objekt "."
 $\rightarrow$ Artikel "Katze jagt" Artikel Substantiv "."
 $\rightarrow$ "die Katze jagt" Artikel Substantiv "."
 $\rightarrow$ "die Katze jagt die" Substantiv "."
 $\rightarrow$ "die Katze jagt die Maus."

Die durch G_1 erzeugte Sprache ist

$L(G_1) = \{ \text{"die Maus jagt die Katze .", "die Maus jagt die Maus .", "die Maus beißt die Katze .", "die Maus beißt die Maus .", "die Katze beißt die Maus .", "die Katze beißt die Katze .", "die Katze jagt die Maus .", "die Katze jagt die Katze ."} \}$.

Maschinen als Sprach-Erkennungssysteme und als Übersetzer

Während Grammatiken *Erzeugungssysteme* für (formale) Sprachen sind, werden (mathematische) Maschinen häufig als *Erkennungssysteme* verwendet. Solche Maschinen realisieren dann eine *charakteristische Funktion*, indem sie bezüglich einer Sprache L entscheiden, ob ein gegebenes Wort w zu L gehört oder nicht.

In der Praxis genügt uns natürlich diese Aussage nur in Ausnahmefällen. Wir wollen vielmehr das in der Sprache formulierte Programm auch verwenden (d.h. seine Semantik in Anspruch nehmen). Dazu brauchen wir nicht nur eine Antwort "ja" oder "nein", sondern (im positiven Fall, wenn das Programm syntaktisch korrekt ist) auch eine Darstellung (in einer anderen Sprache L'), die eine effiziente Weiterbearbeitung gestattet, und möglicherweise schon unmittelbar zur Steuerung des Rechners verwendet werden kann (*Maschinencode*). Wir brauchen also einen *Übersetzer*. Dieser ist ebenfalls durch ein Programm realisiert und gehört zu den wichtigsten Komponenten eines Rechnersystems.

Traditionell unterscheidet man verschiedene Typen von Übersetzern: Ist die Übersetzung im wesentlichen durch fortschreitende Ersetzung von Zeichenreihen möglich, so spricht man von einem *Assembler* (von to assemble = zusammenfügen); sind dagegen kompliziertere Analysen der Eingabe erforderlich, so wird der Übersetzer als *Compiler* bezeichnet (von to compile = zusammentragen).

Sprachen, die mit einem Assembler übersetzt werden können (die sogenannten *Assembler-Sprachen*), sind unvermeidlich primitiv und maschinennah, d.h. sie reflektieren die Strukturen und Eigenschaften der jeweiligen Maschine, deren Maschinencode erzeugt werden soll. Assembler-Programme, die in solchen Sprachen abgefaßt sind, verursachen daher relativ hohen Aufwand, für ihre Erstellung und auch bei späteren Änderungen. Der ungünstigste Fall ist die Übertragung des Programms auf einen ganz anderen Rechner (die *Portierung*), bei der das Programm völlig neu geschrieben werden muß. Darum (und aus anderen Gründen) vermeidet man heute den Einsatz von Assembler-Sprachen und setzt "höhere" Sprachen ein (COBOL, FORTRAN, Pascal, MODULA-2, Ada und viele andere), so daß eine direkte Abhängigkeit von der speziellen Hardware weitgehend beseitigt ist.

1.3 Rechner

Rechner (Computer) sind universell verwendbare Datenverarbeitungsgeräte, die nicht nur - wie die Bezeichnung nahelegt - "rechnen", sondern zur Ausführung *beliebiger* Algorithmen, beispielsweise zur Textverarbeitung, Bilderkennung oder Überwachung von Meßgeräten, einsetzbar sind und "selbständig" arbeiten.

Erste Versuche, solche Maschinen zu konstruieren, haben schon in den vergangenen Jahrhunderten stattgefunden. Sie sind in der Literatur ausführlich beschrieben (vgl. zu diesem Thema allgemein Randell, 1973). Ihr Einfluß auf die Entwicklung der Informatik war allerdings sehr gering. Erst Konrad Zuses Arbeiten in Berlin (Patentanmeldung 1936) und, fast gleichzeitig, aber ohne gegenseitige Beeinflussung, die von Howard Aiken in den USA, schufen die technischen Grundlagen, auf denen etwa ab 1950 die Informatik entstehen konnte. Auch in England gab es solche Aktivitäten, die allerdings wegen der strikten Geheimhaltung - es ging um Dechiffrierung - bis heute nicht voll zu würdigen sind.

Zuses und Aikens Rechner waren durch Lochstreifen *programmgesteuert*, d.h. nicht auf ein festes, mechanisch oder elektrisch fixiertes Programm festgelegt. Durch Einsatz mehrerer Lochstreifenleser konnten sogar Unterprogramme realisiert werden, doch blieben vorerst die Programme (auf Lochstreifen) und die Daten (mechanisch, elektromagnetisch oder später elektronisch gespeichert) strikt getrennt. Der berühmteste Rechner dieser Generation war ENIAC (Electronic Numerical Integrator and Computer), ein Ungetüm aus 19.000 Röhren, mit 30 t Gewicht, einer großen Halle Platzbedarf, einer Leistungsaufnahme von 200 kW und Kosten von ca. 2 Millionen Dollar (im Jahre 1945!)

Der eigentliche Durchbruch zu Rechnern, wie wir sie grundsätzlich bis heute verwenden, war der Nachfolger der ENIAC¹, die EDVAC (Electronic Discrete Variable Arithmetic Computer), in der erstmals für Programm und Daten derselbe Speicher verwendet wurde (vgl. die in 1.1.2 beschriebene Möglichkeit, einer Turing-Maschine als Eingabewort die Tabelle einer anderen Turing-Maschine zur Bearbeitung, Analyse oder Simulation zu geben). Computer der mit der EDVAC realisierten Art bezeichnen wir als *von-Neumann-Rechner*, obwohl grundlegende Konzepte dazu anscheinend von Eckert und Mauchley stammen.

Heutige Rechner profitieren von den faszinierenden Entwicklungen der *Elektronik* in den letzten 50 Jahren. *Ein-Chip-Mikrorechner* zu Beginn der 90er Jahre enthalten viele integrierte Schaltungen in miniaturisierter Form auf einem kleinen Siliziumkri-

¹Rechner sind, vermutlich wegen der "Maschine" als Vulgär-Übersetzung des englischen Wortes "machine", auch im Deutschen üblicherweise weiblich, so daß wir hier von *der* ENIAC usw. sprechen.

stall, eben dem *Chip*. Die *Integration* ermöglicht kleinste Abmessungen (nur noch wenige mm²), hohe Geschwindigkeit (ca. 10⁴ schneller als ENIAC), extreme Zuverlässigkeit (technisch fast fehlerfrei im Vergleich zum anfälligen ENIAC-Rechner), minimale Leistungsaufnahme (≤ 1 W) und einen Preis von weniger als 50 Dollar (bei fallender Tendenz). Trotz dieser Entwicklungssprünge ist bei fast allen, auch heutigen Rechnern die *Architektur* - im Sinne der Verarbeitungslogik der auszuführenden Programme - unverändert geblieben: die von-Neumann-Rechnerarchitektur des EDVAC.

1.3.1 Von-Neumann-Rechnerarchitektur

Der Vorschlag, den von Neumann 1945 als Konzept für die EDVAC machte, enthält eine Gliederung, die sich am biologischen Vorbild orientiert (Informationsverarbeitung des Menschen) und bis jetzt alle gängigen Rechner prägt: Sie bestehen aus den fünf *Funktionseinheiten* (siehe Abb. 1.5) Steuerwerk, Rechenwerk, Speicher, Eingabewerk und Ausgabewerk; die ersten drei davon werden nachfolgend kurz beschrieben.

Die sogenannten *Nicht-von-Neumann-Rechner* sind Gegenstand der Forschung, spielen auf dem Markt aber bisher noch keine wesentliche Rolle. Ein Grund dafür dürfte sein, daß die meisten Methoden, Werkzeuge und Programmiersprachen nicht zuletzt auch die Denkgewohnheiten, fest auf die von-Neumann-Architektur ausgerichtet sind.

Steuerwerk

Das Steuerwerk ist das "Herz" des Rechners. Es hat folgende Aufgaben:

- *Laden* der Befehle (Anweisungen des gerade bearbeiteten Programms) aus dem Speicher in der richtigen Reihenfolge,
- *Decodierung* der Befehle,
- *Interpretation* der Befehle,
- *Versorgung* der an der Ausführung der Befehle beteiligten Funktionseinheiten mit den nötigen Steuersignalen.

Da an der Ausführung eines Befehls das Rechenwerk, der Speicher und die Geräteverwaltung beteiligt sind, ist das Steuerwerk mit all diesen Komponenten verbunden (vgl. Abb. 1.5).

Das *Befehlsregister* enthält den Befehl, der gerade ausgeführt wird. Er besteht aus dem *Operationsteil* und dem *Adreßteil*. Der Operationsteil wird in einem *Decodierer* entschlüsselt. Der Ausgang des Decodierers wählt für jeden möglichen Operationscode genau eine Eingangsleitung der Mikroprogrammeinheit aus.

Das *Befehlszählregister* speichert die Adresse des nächsten auszuführenden Befehls. Es ist mit einer Schaltung verbunden, die den Inhalt des Registers nach der Ausführung eines Befehls um 1 erhöht. Bei *Sprungbefehlen* wird jedoch eine im Adreßteil des Befehls stehende Adresse in das Befehlszählregister kopiert.

Die *Mikroprogrammeneinheit* erzeugt mit Hilfe der decodierten Informationen des Operationscodes eine Folge von Signalen zur Ausführung des Befehls. Die Mikroprogrammeneinheit kann fest verdrahtet und unveränderbar oder programmierbar und variabel gestaltet sein (*Mikroprogrammierung*).

Abb. 1.5 Von-Neumann-Rechnerarchitektur

Rechenwerk

Das Rechenwerk ist die Komponente des Rechners, in der *arithmetische* (z.B. Addition, Subtraktion) und *logische* Verknüpfungen (z.B. und, oder, nicht) durchgeführt werden. Deshalb wird das Rechenwerk auch ALU (Arithmetic Logical Unit) genannt. Die für die Verknüpfungen notwendigen *Operanden* (i.a. zwei) werden dem Rechenwerk vom Steuerwerk zugeführt.

Steuerwerk und Rechenwerk faßt man auch unter der Bezeichnung *Prozessor* (CPU = Central Processing Unit) zusammen. Alle arithmetischen Operatoren (incl. die vier Grundrechenarten) können auf die Basis-Operatoren "Verschieben der Stellen im Register", "Stellenweises Komplementieren ($0 \rightarrow 1, 1 \rightarrow 0$)" und "Addieren" zurückgeführt werden: Subtraktion ist die Addition des Komplements, Multiplikation die wie-

derholte Addition, Division (mit Rest) die wiederholte Subtraktion. Daher sind die grundlegenden Verknüpfungselemente des Rechenwerks *Addierwerk* und *Komplementierer* (siehe Abb. 1.6).

Die Operationen können entweder in serieller oder in paralleler Arbeitsweise durchgeführt werden. Serielle Organisation benötigt nur ein binäres Addierwerk, ist jedoch relativ langsam. Parallele Methoden sind wesentlich schneller, erfordern jedoch viele gleichartige Bausteine und aufwendige Steuermechanismen.

Das Steuerwerk versorgt das Rechenwerk mit den für die richtige Durchführung der arithmetischen Operationen notwendigen *Steuersignalen* (z.B. geordnete Bereitstellung der notwendigen Operanden). Die Algorithmen zur Multiplikation und Division sind ebenfalls im Steuerwerk realisiert.

Abb. 1.6 Aufbau des Rechenwerks

Weitere Einheiten des Rechenwerks sind verschiedene Hilfs- und Zwischen-Register (in denen z.B. die Operanden gespeichert werden) und der *Akkumulator* (der das Ergebnis enthält) sowie eine Operationssteuerung, die dafür sorgt, daß die durch das Steuerwerk veranlaßten Rechenoperationen korrekt ablaufen. Meist enthält das Rechenwerk spezielle Schaltungen, um in speziellen Fällen einzugreifen, z.B. beim Versuch, durch null zu teilen.

Speicher

Der Speicher ist die Rechnerkomponente zum "Aufbewahren" von Daten. *Digitale* Speicher bestehen aus Speicherelementen, die in der Lage sind, abhängig von einem äußeren *Signal* genau einen von mehreren erlaubten *Zuständen* anzunehmen und so lange in ihm zu verweilen, bis er durch ein anderes Signal geändert wird. Heute gibt es praktisch nur *binäre Speicher* mit zwei Zuständen, denen ein binäres Alphabet zugeordnet wird, etwa $\{0,1\}$, $\{L, H\}$ (Low, High für Strom an/aus) oder $\{N, Y\}$ (für No, Yes). Ein solches Speicherelement speichert 1 *Bit* (Zeichen der Einheit: bit).

Da es mit kleinen und großen Buchstaben, Ziffern und einer großen Auswahl von Sonderzeichen fast 200 verschiedene Zeichen gibt, sind 8 bit zur Speicherung eines Zeichens nötig. Daher spielt diese Einheit, das *Byte*, eine besondere Rolle, auch bei der Organisation des Speichers. Ein Byte ist also sowohl eine Gruppe aus acht Elementarspeichern als auch die Einheit der Speicherkapazität. Wie bei Einheiten üblich, verwendet man die Zusätze "k", "M" und "G" für kilo, Mega und Giga (also beispielsweise "Das Programm belegt 190 kByte"). Da aber die Speicher wegen der binären Adressierung nicht dezimal organisiert sind, liegt hier nicht der Faktor 10^3 , sondern 2^{10} zugrunde, also nicht 1.000, sondern 1.024. Damit handelt es sich, genau genommen, bei 1 MByte um 1.048.576, bei einem GByte um 1.073.741.824 Byte.

Die Lokalisierung eines gespeicherten Datums erfolgt durch eine eindeutige Identifikation des Speicherelements, in dem sich das Datum befindet (*Adresse*). I.a. ist nicht jedem einzelnen Speicherelement des Speichers eine Adresse zugeordnet, sondern nur *Gruppen* von Speicherelementen, die alle eine gleiche für den Speicher charakteristische feste Größe haben. Eine solche Gruppe, also eine kleinste adressierbare Einheit eines Speichers, heißt *Speicherzelle*. Eine Speicherzelle entspricht im allgemeinen einem Byte. Die Zusammenfassung mehrerer Speicherzellen (meist 4 oder 8) nennt man *Speicherwort* oder kurz *Wort*.

Den Vorgang, eine Speicherzelle zu lokalisieren und das in ihr gespeicherte Datum abzufragen oder zu verändern, bezeichnet man als *Zugriff*. Man unterscheidet den *lesenden* Zugriff (Lesen), bei dem der Inhalt einer Speicherzelle abgerufen, aber nicht verändert wird, und den *schreibenden* Zugriff (Schreiben), bei dem die Speicherzelle mit einem neuen Inhalt versehen wird. Je nach Bauweise und den damit verbundenen Eigenschaften unterscheidet man Speicher bezüglich der nachfolgenden Kriterien.

Zugriffszeit: Zeit, die zum Lesen bzw. Schreiben einer Speicherzelle benötigt wird. Sie setzt sich zusammen aus der Zeit zur Lokalisierung und Ansteuerung der betreffenden Speicherzelle und der Schaltzeit der Speicherelemente. Die Zugriffszeit liegt je nach Typ des Speichers (Hauptspeicher oder externer Speicher, siehe unten) zwischen wenigen Nanosekunden (Nano = 10^{-9}) und mehreren Sekunden.

Die Hauptspeicher heute üblicher Rechner haben eine Zugriffszeit von 50 bis 500 ns.

Zugriffsart: Speicher klassifiziert man nach der Methode, mit der auf Speicherzellen zugegriffen werden kann. Ist jede Speicherzelle eines Speichers unabhängig von ihrer Position auf die gleiche Weise mit dem gleichen zeitlichen Aufwand erreichbar, so spricht man von Speichern mit *wahlfreiem* oder *direktem* Zugriff (engl. RAM = *Random Access Memory*). Ist der Speicher nur lesbar, so spricht man von einem ROM (engl. *Read Only Memory*).

Speichertyp: Der *Hauptspeicher* eines Rechners ist immer ein Speicher mit *wahlfreiem* Zugriff. *Externe Speicher*, auch *Hintergrundspeicher* oder *Sekundärspeicher* genannt, bei denen die Speicherzellen nur zeitperiodisch erreichbar sind, heißen Speicher mit *zyklischem* Zugriff. Diese im Vergleich zum schnellen *wahlfreien* Zugriff sehr langsame Zugriffsart haben Speicher, bei denen die Speicherzellen rotieren und nur periodisch zugänglich sind, z.B. Magnetplattenspeicher oder Magnettrommelspeicher. Externe Speicher mit *sequenziellem* Zugriff sind solche, bei denen auf eine Speicherzelle erst dann zugegriffen werden kann, wenn auf eine von der Position der Speicherzelle abhängige Anzahl anderer Speicherzellen zunächst zugegriffen wurde. Beispiel für einen Speicher mit sequenziellem Zugriff ist der Magnetbandspeicher.

Zykluszeit: Zeitspanne vom Beginn eines Speichervorgangs bis zu dem Zeitpunkt, an dem ein neuer Speichervorgang beginnen kann. Die Zykluszeit setzt sich zusammen aus der Zugriffszeit (siehe oben) und möglicherweise einer *Regenerationszeit*, in der beim Lesen zerstörte Informationen in den Speicher zurückgeschrieben werden.

Kapazität: Die Speicherkapazität wird bestimmt durch die Anzahl der Speicherzellen, die ein Speicher enthält. Sie wird i.a. in kByte oder MByte gemessen.

Statische oder nichtflüchtige Speicher behalten ihren Inhalt (mindestens solange die Versorgungsspannung eingeschaltet bleibt). *Dynamische* oder flüchtige Speicher verlieren ihre gespeicherten Informationen nach einem bestimmten Zeitintervall, z.B. da die Information mittels eines elektrischen Feldes gespeichert wird, das sich jedoch nach einiger Zeit entlädt (Leckstrom). Daher muß vor Ablauf dieses Zeitintervalls der Inhalt jeder Speicherzelle neu beschrieben werden (*Auffrischung*, engl. refresh).

Von-Neumann-Prinzipien

Nachfolgend sollen die wesentlichen Prinzipien der klassischen Von-Neumann-Rechnerarchitektur noch einmal zusammengefaßt werden:

1. Der Rechner besteht aus fünf Funktionseinheiten: dem Steuerwerk, dem Rechenwerk, dem Speicher, dem Eingabewerk und dem Ausgabewerk.
2. Die Struktur des Von-Neumann-Rechners ist unabhängig von den zu bearbeitenden Problemen. Zur Lösung eines Problems muß von außen das Programm eingegeben und im Speicher abgelegt werden. Ohne dieses Programm ist die Maschine nicht arbeitsfähig.
3. Programme, Daten, Zwischen- und Endergebnisse werden in demselben Speicher abgelegt.
4. Der Speicher ist in gleichgroße Zellen unterteilt, die fortlaufend durchnummeriert sind. Über die Nummer (Adresse) einer Speicherzelle kann deren Inhalt abgerufen oder verändert werden.
5. Aufeinanderfolgende Befehle eines Programms werden in aufeinanderfolgenden Speicherzellen abgelegt. Das Ansprechen des nächsten Befehls geschieht vom Steuerwerk aus durch Erhöhen der Befehlsadresse um Eins.
6. Durch Sprungbefehle kann von der Bearbeitung der Befehle in der gespeicherten Reihenfolge abgewichen werden.
7. Es gibt zumindest
 - arithmetische Befehle wie Addieren, Multiplizieren usw.;
 - logische Befehle wie Vergleiche, logisches nicht, und, oder usw.;
 - Transportbefehle, z.B. vom Speicher zum Rechenwerk und für die Ein-/Ausgabe;
 - bedingte Sprünge.Weitere Befehle wie Schieben, Unterbrechen, Warten usw. kommen hinzu.
8. Alle Daten (Befehle, Adressen usw.) werden binär codiert. Geeignete Schaltwerke im Steuerwerk und an anderen Stellen sorgen für die richtige Entschlüsselung (Decodierung).

Die Architektur des Von-Neumann-Rechners fällt in die Klasse der sogenannten SISD (single instruction stream, single data)-Architekturen. Diese Architekturen sind gekennzeichnet durch

- *einen* Prozessor (bestehend aus Steuer- und Rechenwerk) und
- die Erzeugung *einer* Befehls- und *einer* Operandenfolge mit streng sequentieller Abarbeitung.

Alternative Rechnerarchitekturen

Alternative Rechnerarchitekturen, die erst am Beginn eines breiteren kommerziellen Einsatzes stehen, verfügen über mehr Prozessoren. Durch *Nebenläufigkeit* oder *Parallelität* sollen Algorithmen effizienter, d.h. schneller ablaufend und/oder weniger Platz beanspruchend ausführbar sein.

Z.B. kann man die übliche Matrix-Multiplikation (jedes Element einer Ausgangsmatrix geht mehrfach in Multiplikationen mit Elementen der anderen Ausgangsmatrix ein) durch weitgehend paralleles, stellenweises Multiplizieren mit "verschränktem" Addieren von Zeilen- bzw. Spaltenelementen beschleunigen. Auch das Suchen eines Elements in einer großen Datenmenge durch paralleles Suchen in disjunkten Teilmengen mit Meldungspflicht beim Finden des Elements ist ein anschauliches Beispiel für Effizienzsteigerung durch paralleles Nutzen mehrerer Prozessoren.

Je nachdem, ob die Prozessoren gleichzeitig den gleichen oder unterschiedliche Befehle auf den gleichen oder verschiedenen Datenströmen ausführen können, werden die resultierenden Architekturen weiter differenziert.

1.3.2 Rechnersysteme

Rechnersysteme bestehen aus physikalischen (*Hardware*) und veränderbaren, durch Programme gegebene (*Software*) Komponenten. Neben dem eigentlichen, in der von-Neumann-Architektur bereits vorgestellten Rechner gehören zur Hardware eines Rechnersystems *periphere Geräte* zur Datenerfassung, -speicherung, -ausgabe und -übertragung:

- zur Erfassung z.B. Lochkarten- und Lochstreifenleser (zunehmend antiquiert), Tastaturen, Klarschrift- oder Markierungsleser, Scanner, berührungs- oder maus-sensitive Datensichtgeräte/Terminals;
- zur Speicherung als Ergänzung zum Hauptspeicher z.B. Magnetbänder, -platten und -trommeln, aber auch optische Speichermedien wie CD-ROM und Bildplatten;
- zur Ausgabe z.B. Drucker, Plotter (Zeichengerät), Datensichtgeräte.

Bei der Datenübertragung unterscheidet man die internen, im Rechnersystem stattfindenden Datenübertragungen zwischen den gerade aufgezählten Geräten und die externen, das Rechnersystem in ein *Rechnernetz* verlassenden Datenübertragungen (weiter unten in diesem Abschnitt beschrieben).

Betriebssystem

Bei der Verwendung eines Rechners sind gewisse Aufgaben immer wieder zu lösen, beispielsweise die Speicherung einer Datei, der Dialog über die Peripheriegeräte und das Binden und Ausführen eines Programmsystems. Daher sind Programme für solche und ähnliche Zwecke im *Betriebssystem* zusammengefaßt, das in der Regel zur Grundausstattung des Rechners gehört.

Ein Betriebssystem nimmt den Benutzern Routineaufgaben ab, regelt das korrekte Zusammenwirken der physikalischen Einheiten (vor allem Speicher, Drucker, Bildschirme, Tastaturen, Netzanschlüsse) und steuert die Bearbeitung der Programme. Ordnete man früher dem Betriebssystem nur die elementarsten Aufgaben zu (Prozeß- und Speicherverwaltung), so umfaßt der Begriff heute auch viele Funktionen mit

Werkzeug-Charakter, beispielsweise Übersetzer, Backup-Routinen (zur Sicherung der Programme und Daten) oder Monitore zur Überwachung der Rechnerauslastung. Abb. 1.7 zeigt den typischen Aufbau eines Betriebssystems.

Abb. 1.7 Aufbau eines Betriebssystems

Die *Organisationsprogramme* verwalten die Speicher, die Prozessoren und die Geräte einschließlich der Netze. Diese Teile des Betriebssystems steuern somit insbesondere

- die Zuteilung des Prozessors an die zu bearbeitenden Prozesse (Programme),
- die Verwaltung der auf Bearbeitung wartenden Prozesse,
- alle an den Rechner angeschlossenen Speicher und die Zuteilung von Speicherbereichen an die einzelnen Benutzer bzw. Programme,
- die Nutzung des Hauptspeichers (Teile des Hauptspeichers werden auf externe Speicher ausgelagert und später für erneute Zugriffe wieder eingelagert; für eine gute Ausnutzung des Rechnersystems sind Nachlade- und Auswahl-Strategien von zentraler Bedeutung),
- die Art und Weise, wie Daten auf den Speichern abgelegt werden und wie man erneut auf sie zugreifen kann,
- die Auswahl und die Zuweisung von Ein- und Ausgabegeräten,
- die Überwachung der Datenübertragung zwischen Geräten und Rechnern, z.B. die Ermittlung der Zielrechner und Datenbanken für Anfragen, elektronische Mitteilungen usw.,
- die Überprüfung von Sicherungsmaßnahmen und Zugriffsberechtigungen entsprechend der Vorgaben durch den Benutzer oder den Systemverwalter.

Diese Aufzählung macht deutlich, wieviele stark von einander abhängige Prozesse auf verschiedenen Einheiten korrekt ablaufen müssen. Aus Sicht der Informatik ist ein Prozeß der Verarbeitungsvorgang, den eine einzelne Verarbeitungseinheit (ein "Prozessor") durchführen kann. Abläufe in der Realität können durch viele miteinander kommunizierende Prozesse beschrieben werden. *Müssen* diese Prozesse zur

gleichen Zeit nebeneinander ablaufen, so spricht man von *parallelen Prozessen*; sind die Prozesse unabhängig voneinander und können sie gleichzeitig oder in beliebiger Reihenfolge nacheinander durchgeführt werden, so spricht man von *Nebenläufigkeit*.

Menschliche Organisationen bestehen meist aus vielen Prozessen, die aufeinander abgestimmt sind (die sich "*synchronisieren*") oder die unabhängig voneinander (also "*asynchron*") ablaufen. Wenn eine an einem Tisch sitzende Gruppe von Menschen in einem Restaurant individuell verschiedene Essen bestellt, so wird die Küche für jede Bestellung einen "Prozeß starten", um (parallel) das jeweilige Essen herzustellen; diese Prozesse werden so koordiniert ("*synchronisiert*"), daß alle Gäste der Gruppe ihr Essen etwa gleichzeitig bekommen. An verschiedenen Tischen bestellte Essen werden dagegen nebenläufig von der Küche bearbeitet (d.h. parallel und/oder sequentiell).

Ein Rechnersystem verwaltet eine Menge von Prozessen, und die Befehle auf der Ebene des Betriebssystems beziehen sich meist auf Prozesse. Für diese müssen Sprachelemente vorhanden sein, die in einfachen Sprachen fehlen, z.B. für das Aufspalten, das Synchronisieren, das Unterbrechen und das Kommunizieren von Prozessen.

Zu den *Programmiersprachen-Systemen* (siehe Abb. 1.7) gehören Compiler und Interpreter, die Programme einer Sprache "bedeutungsäquivalent" in eine andere Sprache transformieren. Programmiersprachen dienen der Darstellung von Algorithmen, Prozessen und der zugehörigen Datenstrukturen. Wenn zwei in Syntax und Semantik vollständig definierte Sprachen S und S' vorliegen, die gewissen formalen Bedingungen genügen, dann kann man ein Programm schreiben, welches (*Quell-*)Programme der Sprache S als Eingabe erhält und semantisch gleichwertige (*Ziel-*)Programme der Sprache S' als Ausgabe liefert (vgl. 1.2.2). Solch ein Programm nennt man einen *Übersetzer*. Sind die Strukturen der Zielsprache S' wesentlich einfacher als die der Quellsprache S, so spricht man von einem *Compiler*, liegt der umgekehrte Fall vor, von einem *Decompiler*. Sind die Strukturen der beiden Sprachen verschieden, aber etwa gleich komplex, so spricht man von einem *Querübersetzer*.

Compiler arbeiten meist nach folgendem Schema:

1. Schritt: *Lexikalische Analyse*.

Die einzelnen Grundbausteine der Sprache werden im Quellprogramm erkannt (z.B. "SteuerSatz" als Bezeichner, das Zuweisungszeichen "!=" aus der Folge der zwei Zeichen ":" und "=".)

2. Schritt: *Syntaktische Analyse*.

Die syntaktische Struktur des Programms wird ermittelt und als *Ableitungsbaum* (Ergebnis der Anwendung kontextfreier Produktionen der zugrundeliegenden Grammatik) dargestellt. Zugleich wird eine *Symboltabelle* erstellt, in der alle auftretenden Bezeichner mit ihren Bedeutungen abgelegt werden.

3. Schritt: *Semantische Analyse* und *Code-Generierung*.

Der Ableitungsbaum wird durchlaufen und in das Zielprogramm umgewandelt.

4. Schritt: *Code-Optimierung*.

Das erzeugte Programm wird hinsichtlich seiner späteren Laufzeit und des Speicherbedarfs verbessert.

Bei einigen Programmiersprachen ist es möglich oder sogar vorteilhaft, nur jeweils den gerade auszuführenden Befehl zu übersetzen (ihn zu "interpretieren"). Ein System, das so arbeitet, wird als *Interpreter* bezeichnet. Dadurch wird zwar die Programm-Ausführung wesentlich langsamer, doch entfällt dafür die Wartezeit vorher, die für die Compilierung erforderlich wäre. Auch kann die Speicherverwaltung zur Laufzeit wesentlich flexibler als zur Übersetzungszeit gehandhabt werden.

In Programmiersprachen ist die Abgrenzung zwischen Compilern und Interpretern nicht so scharf, wie die Definitionen oben vermuten lassen. So gibt es kaum einen Interpreter, der nicht wenigstens die elementarsten Vereinfachungen vorab durchführt (im Sinne eines Scanners). Andererseits überläßt auch ein Compiler gewisse Standardaufgaben der interpretativen Bearbeitung, beispielsweise die Ein- und Ausgabe oder die Behandlung von Laufzeitfehlern. Dazu dient das dem Compiler zugeordnete *Laufzeitsystem*.

Die *Dienstprogramme* (siehe Abb. 1.7) übernehmen Verwaltungs- und Routineaufgaben, die in den unterschiedlichsten Anwendungen vorkommen, z.B.

- Erstellung und Veränderung (*Edieren*) von Texten, Graphiken und Programmen (s.u.),
- Laden von Programmen,
- Sortieren von Datenbeständen,
- Verwaltung von Dateien (bis hin zu einfachen Datenbanken, s.u.),
- schrittweise Ausführung von Programmen mit Anzeige des aktuellen Programmzustands (für die Fehlersuche, *Debugging*),
- Kommunikation mit anderen Benutzern, auch auf weit entfernten Rechnern (Electronic Mail).

Wer mit einem Computer arbeitet, lernt in der Regel neben einigen Befehlen des Betriebssystems als erstes einen *Editor* kennen, denn Texte, Graphiken, Bilder und sonstige Dokumente müssen erstellt, gelöscht, manipuliert und übertragen werden, und genau dies unterstützt bzw. leistet der Editor. Suchen, Ersetzen, Einkopieren, Sortieren, Umnummerieren, usw. sind zusätzliche Funktionen, die in der Regel geboten werden.

Editoren können die Arbeit wesentlich erleichtern, sie können zum Beispiel die syntaktische Struktur von Texten und Programmen überprüfen, gewisse Vorgehensweisen erzwingen, Tests auf Plausibilität und Vollständigkeit durchführen, Schriftgrößen und Schriftarten festlegen und vieles mehr. Editoren sind sehr umfangreiche

Programme, da sie neben den vielen Algorithmen und den syntaktischen Überprüfungen zugleich über Schnittstellen zur Speicherverwaltung sowohl des Hauptspeichers als auch (unter Benutzung des Betriebssystems) des externen Speichers verfügen müssen. Früher arbeiteten sie "zeilenorientiert", d.h. es wurde jeweils genau eine Zeile (bis zu 80 Zeichen, abgeschlossen durch ein Zeilenende-Zeichen) verarbeitet und in dieser Form abgespeichert.

Moderne Editoren sind bildschirm- und syntaxorientiert, bringen also Text und/oder Graphiken in Ausschnitten ("*Fenstern*") auf den Bildschirm und bieten dem Benutzer einen komfortablen Dialog zur Arbeit am Dokument, erkennen zugleich Syntaxfehler und schlagen Korrekturen vor.

Akten, Dokumente, Zeichnungen usw., kurz Daten, legt man in geeigneten "Behältern" wie Ordnern, Heften, Alben, Mappen usw. ab, ohne sich um deren technische Details (wie Heftung, Mechanismus u.ä.) zu kümmern. Auch wenn man mit Rechnern arbeitet, möchte man sich um die Art und Weise, wie Daten abgelegt werden, nicht kümmern. Wichtig ist nur, daß sie nicht von anderen Personen oder gar automatisch verändert werden, daß man sie wiederfinden und daß man sie erzeugen, verändern oder löschen kann.

Ein Informationssystem oder eine *Datenbank* bietet dem Benutzer die Möglichkeit, eigene Datentypen zu deklarieren, Stichwörter und Überschriften (für Zwecke des Wiederfindens) anzugeben und nach dieser Vereinbarung beliebig viele Daten zu speichern; zugleich muß die Datenbank (DB) über ein *Verwaltungssystem* verfügen, das die Daten ablegt, die Zugriffsrechte überwacht, den Zugriff steuert und Anfragen bearbeitet.

Hierbei müssen verschiedenen Benutzern verschiedene Ausschnitte ("Sichten") des Datenbestands vermittelt werden. In einer DB, die alle Informationen über die Studierenden einer Hochschule enthält, darf die Immatrikulationsstelle auf alle personen- und studienbezogenen Daten zugreifen, sie erhält also eine fast vollständige Sicht auf die Daten (mit Ausnahme der Prüfungsdaten); die Geschäftsstelle eines Fachbereichs erhält nur Zugriff auf allgemeine Daten (z.B. Adressen) der Studierenden des jeweiligen Fachs; das Prüfungsamt darf Adressen und prüfungsrelevante Daten lesen und verändern usw. Die DB umfaßt somit einen großen Datenbestand, auf den jede Organisationseinheit nur in gewissen Sichten zugreifen darf.

Ähnliches gilt für verschiedene Anwendungsprogramme, die alle auf die gleiche DB zugreifen. Das steuernde DB-Verwaltungsprogramm muß Funktionen zur Definition der Berechtigungen und Sichten sowie zum Neu-Anlegen und Löschen, Prüfen und Ändern, Suchen und Bereitstellen, Sortieren, Kopieren usw. enthalten.

Künftige DB-Systeme werden auch sehr umfangreiche und unterschiedlich strukturierte, sogenannte "komplexe" Objekte verwalten müssen. Die hierin enthaltenen Daten müssen automatisch auf innere Widersprüche überprüft werden, es sind sehr komplexe und zeitaufwendige Verarbeitungen durchzuführen, und es müssen Versionen

gespeichert werden, auf denen man evtl. wieder aufsetzen will. Solche Anwendungen, sogenannte *Nicht-Standard-DB-Anwendungen* treten z.B. im Büro, im Software Engineering, in Expertensystemen und in technischen Disziplinen auf.

Einbettung der Programmierung

Programmieren ist eine Tätigkeit, bei der man nicht ohne *ingenieurmäßige* Techniken und Werkzeuge auskommt (oder auskommen sollte). Man arbeitet in der Regel in einer "*Programmierungsumgebung*", die Unterstützung leistet beim Schreiben von Texten, Zeichnungen und Programmen (Editoren), beim Archivieren, Wiederfinden, Einbinden oder Sortieren (Datenbank- oder Betriebssystemfunktionen), beim Übertragen in den Maschinencode oder andere Sprachen (Übersetzer).

Dabei ist zu bedenken, daß die Codierung nur einen kleinen Ausschnitt der Aktivitäten darstellt, die zur Software-Entwicklung gehören (ca. 20 % der Kosten). Bevor die Codierung beginnen kann, wurde durch Analysen, Spezifikationen und Entwürfe ein Konzept geschaffen; wenn sie beendet ist, muß die Software getestet, integriert und in u.U. vielen Versionen und Varianten verwaltet werden. Parallel müssen Entwickler und Benutzer geschult werden. Schließlich beginnt die teuerste Phase, die sogenannte *Wartung*, also der Einsatz mit meist vielen, unvorhergesehenen Modifikationen, Erweiterungen und Korrekturen. Diese Themen sind Gegenstand des Software Engineering. Einige Aspekte (Spezifikation, Verifikation, Test) werden wir in Kapitel 3 behandeln.

Rechnernetze

Unter einem *Rechnernetz* versteht man die Kopplung (Vernetzung) mehrerer Rechner, die möglicherweise in ihrer Rechnerarchitektur und/oder ihrem Betriebssystem unterschiedlich sind und i.a. räumlich getrennt stehen. Die Vernetzung kann u.a. aus folgenden Gründen sinnvoll sein:

Datenverbund: Nutzung von Datenbeständen, die auf einzelne Rechner des Netzes verstreut sind.

Beispiel: Kopplung von Bibliotheksrechnern, um die auf den einzelnen Rechnern gespeicherten Literaturangaben gemeinsam zu nutzen.

Betriebsmittelverbund: Nutzung teurer Software- und Hardwarebetriebsmittel, die nicht auf jedem Rechner des Netzes bereit gehalten werden können.

Beispiele solcher Betriebsmittel sind Spezialrechner hoher Rechengeschwindigkeit, Bilderfassungsgeräte, Übersetzer für bestimmte Programmiersprachen, Spezialsoftware für Informationssysteme usw.

Lastverbund: Gleichmäßige Verteilung der benötigten Rechenleistung auf die an das Netz angeschlossenen Rechenanlagen. Fällt ein Rechner aus, können andere dessen Aufgabe übernehmen. Der Lastverbund wird sehr erleichtert, wenn die einzelnen Rechner *kompatibel* sind, d.h. daß an das Rechnernetz gegebene

Aufträge auf jedem beteiligten Rechner in gleicher Weise ausgeführt werden können. Rechnernetze mit dieser Eigenschaft heißen *homogen*, die übrigen *heterogen*.

Man unterscheidet Rechnernetze, deren *Knoten* (Rechner) über große Entfernungen unter Verwendung meist *öffentlicher* Netze verbunden sind (engl. WAN = Wide Area Network), und solche, deren Knotenrechner über kurze Entfernungen bis 2,5 km unter Verwendung *privater* Verbindungsleitungen verbunden sind (engl. LAN = Local Area Network).

Abb. 1.8 Vernetzungstopologien

Als *Topologie* eines Rechnernetzes bezeichnet man die Form der Verteilung der Knotenrechner und deren Vernetzungsstruktur. Wichtige Topologien sind - vgl. Abb. 1.8, in der dunkle Knoten leistungsstarke Zentralrechner (*server*), helle Punkte normale Arbeitsstationen (*clients*) symbolisieren - die *sternförmige* (a), die *dezentrale* (b), die *verteilte* (c) und die *hierarchische* Topologie (d). Relativ unabhängige Arbeitsplatzrechner werden meist entweder durch einen Bus (e), z.B. Ethernet oder einen Ring (f), z.B. Tokenring miteinander verbunden. Die Regeln, nach denen am Rechnernetz beteiligte Rechenanlagen Kommunikationsverbindungen auf- und abbauen, Informationen austauschen usw., werden in *Kommunikationsprotokollen* festgelegt.

Bemerkung: An dieser Stelle empfiehlt es sich, die lokale Rechnersituation und -vernetzung in seiner unmittelbaren Umgebung anzusehen, bevor im nächsten Kapitel die Praxis des Programmierens und damit die Nutzung eines Rechnersystems beginnt.

Die wichtigste Kenngröße der Rechnervernetzung ist (natürlich nach ihrer Zuverlässigkeit) ihre Kapazität, d.h. der maximale Informationsfluß (Einheit bit/s). Dabei geht es in der Praxis darum, zwischen den hohen Kosten einer breitbandigen Leitung und der niedrigen Übertragungsrate auf einer schmalbandigen Leitung einen der Aufgabe angemessenen Kompromiß zu finden.

1.4 Informatik als Wissenschaft

Die Informatik als *Wissenschaft*, d.h. unter dem Gesichtspunkt *universitärer* Lehre und Forschung, gliedert sich in vier Säulen: die Theoretische, die Praktische, die Technische und die Angewandte Informatik. Theoretische, Praktische und Technische Informatik sowie ingenieurmäßige Anteile der Angewandten Informatik faßt man oft unter dem Oberbegriff "*Kerninformatik*" zusammen.

Theoretische Informatik

Sowohl für die Formulierung und Untersuchung von Algorithmen als auch für die Rechnerkonstruktion spielen Methoden und Modelle aus der Mathematik eine wesentliche Rolle. Da die Struktur von Computern ständig komplexer wird, nimmt auch der *Abstraktionsgrad* einer angemessenen Beschreibung zu. Für die Untersuchung von Fragestellungen in diesem Bereich sind gute Kenntnisse der "*strukturellen*" *Mathematik* nötig, die eine Reihe formaler Methoden zur Beschreibung von Systemen bietet. Bei der Untersuchung geeigneter Sprachen spielen Methoden der Logik eine wichtige Rolle.

Beispiele für Teilgebiete der Theoretischen Informatik sind Formale Sprachen, Theorie der Netze und Prozesse, Automatentheorie, Semantik und Komplexitätstheorie.

Praktische Informatik

Prinzipiell wäre es möglich, jeden beliebig komplexen Algorithmus direkt in der Sprache eines bestimmten Rechners zu formulieren; der Aufwand wäre allerdings gigantisch. Um mit begrenztem Aufwand zu zweckmäßigen Programmen zu kommen, müssen wir den Rechner auch für die Arbeit der Programmentwicklung als komfortables *Werkzeug* einsetzen und für regelmäßig wiederkehrende Probleme Standardlösungen schaffen. Programme, die in einer uns Menschen vorteilhaften Notation formuliert sind, müssen von Compilern oder Assemblern in eine dem Rechner verständliche und ausführbare Form übertragen werden. Ein Betriebssystem überwacht die Ausführung der Programme, die bei größeren Rechenanlagen oftmals gleichzeitig ablaufen, und übernimmt die Steuerung der Ein- und Ausgabe. Informationssysteme und Datenbanken verwalten umfangreiche Datenbestände, weitere Programmsysteme, sogenannte *CASE-Tools* unterstützen die Software-Bearbeitung im weiteren Sinne, also die Spezifikation, den Entwurf, den Test, die Versionenverwaltung usw.

Bildverarbeitende Systeme, Robotersteuerung, Spracherkennung und *Expertensysteme* ("Ratgeber-Programme", die ähnlich Experten auf eng begrenztem Gebiet aus Fakten Schlüsse ziehen) sind weitere, an Bedeutung gewinnende Themen der Praktischen Informatik. Software Engineering bezeichnet die Disziplin, die sich mit den Methoden, Notationen und Werkzeugen der Software-Bearbeitung allgemein befaßt.

Technische Informatik

In der Technischen Informatik befaßt man sich mit dem funktionellen Aufbau von Computern und den zugehörigen Geräten sowie mit dem logischen Entwurf und der konkreten Entwicklung von Rechnern, Geräten und Schaltungen (Hardware). Die Schnittstelle zu Betriebssystemen und die Zusammenstellung von Computern spielen eine wichtige Rolle.

Beispiele für Teilgebiete der Technischen Informatik sind Rechnerarchitektur, Prozeßdatenverarbeitung, Fehlertoleranz, Leistungsmessung und VLSI-Entwurf (Very Large Scale Integration), d.h. die Integration sehr vieler Funktionen auf *einem* Chip, also in *einem* Mikroprozessor.

Angewandte Informatik

Informatik versteht sich als anwendbare, sogar nach Anwendung verlangende Wissenschaft, die viele Sparten des Lebens beeinflußt. Unter Angewandter Informatik faßt man die Anwendung von Methoden der Kerninformatik in anderen Wissenschaften und die Entwicklung spezieller Verfahren und Darstellungstechniken zusammen.

Die Angewandte Informatik untersucht Abläufe in den unterschiedlichsten Bereichen auf ihre Automatisierbarkeit durch Computer. Im Vordergrund steht dabei das ingenieurmäßige Vorgehen bei der Entwicklung von Software, d.h. von Programmsystemen, die solche Anwendungsfälle abdecken sollen. Gewisse, aus verschiedenen Anwendungen stammende gemeinsame Gebiete sind *ergonomische* Fragen und Probleme der *Mensch-Maschine-Kommunikation*. Dies überschneidet sich auch mit Forschungsgebieten aus der Psychologie (Kognitionswissenschaften), aus der Pädagogik (Training und Erklärung), aus den Sozial- und Wirtschaftswissenschaften (Arbeitsstrukturen, Organisationsfragen, Führungssysteme) usw.

Beispiele für Teilgebiete der Angewandten Informatik sind die Wirtschaftsinformatik, die Rechtsinformatik und die Medizininformatik. Solche Gebiete werden auch als "Bindestrich-Informatik" bezeichnet. Die Grenze zwischen Angewandter und Praktischer Informatik ist fließend.

Mitunter empfiehlt es sich, zwischen Angewandter Informatik (Methoden und Verfahren der Kerninformatik unter anwendungsorientierten Aspekten) und *Anwendungen der Informatik* (Fragen des Einsatzes von Informatik-Werkzeugen in verschiedenen Anwendungsgebieten) zu unterscheiden.

Die Beziehungen zwischen den Teilgebieten der Informatik

Abb. 1.9 zeigt einen Versuch, die vier vorgestellten Teilgebiete der Informatik in einer "*Schichtenarchitektur*" (zwölf Schichten im grau unterlegten Kasten im Zentrum der

Abb. 1.9) anzuordnen und mit *Grundlagenwissenschaften* (unterhalb) und *Anwendungsgebieten* (oberhalb des Kastens) in Beziehung zu setzen.

Abb. 1.9 Schichtenarchitektur und Beziehungen der Informatik

Die Schichtenarchitektur ist so zu lesen, daß jeweils die "Produkte" einer Schicht, also ihre Methoden, Verfahren und Ergebnisse, zur Lösung von Aufgaben der darüber liegenden Schicht genutzt werden können.

Als Ingenieurwissenschaft zielt die Informatik primär auf die Anwendung ihrer Resultate auf allen Gebieten, also in der Wirtschaft, Technik, Verwaltung und Ausbildung, eventuell auch in der Kunst. Hier, in der obersten Ebene des Bildes, stehen *Anwendungssysteme* im Vordergrund. Durch den stark anwachsenden Rechnereinsatz auch auf Gebieten wie Psychologie, Recht, Sozialwissenschaften usw. ergeben sich viele Querbezüge, was die Entwicklung der Informatik zu einer *interdisziplinären*, zunehmend Grundlagencharakter erhaltenden Wissenschaft unterstreicht.

Die auf der 2. Funktionsebene der Schichtenarchitektur angesiedelten *Entwicklungssysteme* sollen als weitgehend anwendungsneutrale Softwarewerkzeuge die Erstellung der gerade skizzierten Anwendungssysteme erlauben. Dabei stützt sich diese Ebene selbst aber auf *Sprachsysteme* (3. Funktionsebene) ab, die aus Werkzeugen wie Editoren, Übersetzer, Tester usw. bestehen, die sich der Werkzeuge auf der 4. Funktionsebene "*Systemsoftware*" bedienen usw.

Am unteren Ende trifft man schließlich auf elektronische Bauelemente, die durch physikalische Vorgänge beschreibbar sind und z.B. auf Erkenntnissen der Grundlagenwissenschaft "Physik" basieren.

In der Schichtenarchitektur lassen sich die vier Teilgebiete der Informatik wie in Abb. 1.9 dargestellt anordnen:

- die Technische Informatik umfaßt etwa den Bereich von elektronischen Bauelementen bis zu Fragestellungen der Rechnerarchitekturen,
- die Praktische Informatik setzt dort etwa an und betrachtet alle darüber liegenden Funktionsebenen mindestens einschließlich der Entwicklungssysteme,
- die Angewandte Informatik deckt ungefähr die obersten drei Ebenen mit dem Schwerpunkt Anwendungssysteme ab, während
- die Theoretische Informatik als Grundlage und Klammer der drei übrigen Gebiete die formalen Hilfsmittel bereitstellt und sich dabei auf Teile der Mathematik abstützt, aber auch Beziehungen zur Philosophie aufweist.

Ein wichtiger Aspekt ist die *Ausbildung in Informatik* (z.B. im schulischen, beruflichen und universitären Bereich), die über Fragen der *Didaktik* mit Aspekten der Pädagogik in Berührung kommt (rechts neben dem Kasten in Abb. 1.9). Allgemein ist zu beachten, daß die Abb. 1.9 technische Systeme in den Vordergrund stellt, zu denen aber jeweils auch alle Aspekte der Methodik und Notation zu ergänzen sind.

Insgesamt zeigt Abb. 1.9, daß die Informatik längst nicht mehr wie in den 50er und 60er Jahren eine Sammlung von Spezialgebieten anderer Wissenschaften, vor allem der Mathematik, Logik und Elektrotechnik ist, sondern sich zu einer zusammenhängenden, eigenständigen und theoretisch fundierten Disziplin entwickelt hat. Da sie

sich immer stärker in immer mehr Bereichen auswirkt, bekommt sie zunehmend den Charakter einer zentralen *Grundlagenwissenschaft*.

Auswirkungen der Informatik in der Gesellschaft

Der Einsatz von Computern beeinflusst die Arbeitswelt und den Freizeitbereich nachhaltig und betrifft damit die meisten Menschen direkt und sichtbar. Ähnlich wie vor zweihundert Jahren die Dampfmaschine wird auch der Computer als Instrument der Rationalisierung eingesetzt, woraus sich für die Betroffenen oft schwerwiegende *soziale Folgen* ergeben (Abbau oder Wandel von Arbeitsplätzen, Änderungen der beruflichen Anforderungen).

Betrachten wir den Begriff und die Auswirkungen der *Rationalisierung* näher. Im Rahmen der Informatik versteht man unter Rationalisierung den Ersatz menschlicher Einwirkung und Kontrolle durch den Einsatz technischer, vor allem computergesteuerter Maschinen und die Übertragung geistiger Routinearbeit auf Datenverarbeitungsanlagen.

Die Rationalisierung betrifft nicht alle Bereiche der Arbeitswelt in gleichem Maße. Tätigkeiten, die aus einfachen, ständig zu wiederholenden Handgriffen zusammengesetzt sind (z.B. Fließbandarbeit), sind oft besonders einfach durch Maschinen zu leisten. Arbeiten, die in hohem Maße künstlerische oder handwerkliche Fähigkeiten erfordern, Spezialwissen und Erfahrung voraussetzen oder von der menschlichen Person (Sprache, Auftreten) geprägt sind, können dagegen bisher kaum auf Maschinen übertragen werden. Allerdings sind auch handwerkliche Spezialisten nicht davor geschützt, durch eine von der Informatik ausgelöste Strukturkrise getroffen zu werden (wie die Uhrmacher, als die elektronischen Uhren aufkamen).

Beispiele für *Berufszweige*, in denen in den letzten Jahren erhebliche Rationalisierungen stattgefunden haben, sind

Fabrikation: Autos werden heute weitgehend automatisch gefertigt. Die Herstellung und Montage vieler Teile erledigen Handhabungsautomaten (Roboter), deren Bedienung und Wartung nur noch wenige (und andere) Arbeitskräfte benötigt. Schwere oder gesundheitsschädliche Tätigkeiten (z.B. "Über-Kopf-Arbeit", Lackierungsarbeiten) werden bevorzugt auf Maschinen übertragen.

Büro: Anstelle von Schreibmaschinen werden Textautomaten verwendet, die vorgefertigte Texte und Textbausteine gespeichert haben und eine höhere Schreibleistung erlauben (und damit natürlich auch von den Betroffenen *verlangen*). Post wird in Einzelfällen betriebsintern und nach außerhalb nur noch auf elektronischem Wege verschickt; trotzdem scheint das *papierlose Büro* keineswegs näherzurücken, denn die Möglichkeit, mit Hilfe der Textverarbeitung in kurzer Zeit

sehr viel Text zu generieren (wenn auch nicht entsprechend viel Aussage), hat im Gegenteil zu einer steigenden Papierflut geführt.

Druckgewerbe: Elektronische Satzverfahren machten den Beruf des Setzers entbehrlich. Redakteure oder Schreibkräfte tippen die Artikel direkt in den Satz-Computer, der automatisch das Druckbild errechnet, schnelle Korrekturen erlaubt und viele Routinearbeiten (z.B. Durchschuß) übernimmt. Offensichtliche Nachteile, vor allem die oft amateurhafte Gestaltung und fehlende Kontrolle (Orthografie, Interpunktion, Trennungen), werden wegen der wesentlich niedrigeren Kosten und kürzeren Bearbeitungszeiten in Kauf genommen. Der Computereinsatz führt also teilweise zu einem Qualitätsverfall.

Die Rationalisierung von Betrieben ist oft von starken Änderungen der Arbeitsbedingungen und der (Arbeits-) Organisation begleitet. Der Konkurrenzdruck läßt aber in vielen Fällen gar keine Wahl.

Weniger offensichtlich, aber wohl kaum geringer sind die indirekten Auswirkungen der Informatik, beispielsweise durch schnellere Verfügbarkeit wichtiger Informationen für ganz bestimmte Personen oder Gruppen. Hier zeigen sich einerseits die beeindruckenden Möglichkeiten, die uns und anderen der Computer durch Speicherung und Verknüpfungsfähigkeit schafft, verbunden mit Zuverlässigkeit und Verarbeitungsgeschwindigkeit, andererseits die *Gefahren*, die sich aus der schnellen Verfügbarkeit *personenbezogener* Daten und der Konzentration von Informationen in Datenbanken ergeben.

Diese *Datenschutz-Problematik* läßt sich etwa durch die beiden Standpunkte skizzieren "Wenn wir alle vorhandenen Informationen verfügbar machen, können wir den Terrorismus viel wirksamer bekämpfen" gegen "Ich will selbst bestimmen, wer meine Daten bekommt, denn wer Informationen über mich hat, kann sie mißbrauchen". Hier prallen Welten aufeinander, und der Informatiker kann als *Fachmann* nur Auskunft über die technischen Möglichkeiten der Maschine geben; als *Mensch* sollte er sich allerdings - wie jeder andere auch - darüber Rechenschaft ablegen, wie sich sein Handeln oder Stillhalten auf die Welt auswirkt.

2. Imperative Programmierung - die Sprache MODULA-2

Ziel dieses Kapitels ist, die für das Programmieren in einer imperativen, d.h. *befehls-* oder *anweisungsorientierten* Programmiersprache wie MODULA-2 notwendigen Grundlagen zu erarbeiten und wesentliche Sprachkonstrukte kennenzulernen.

Darauf aufbauend sollen später Aspekte der Programmiermethodik und Softwareentwicklung wie Spezifikation und Verifikation von Programmen, aber auch Abstraktion und Datenkapselung eingeführt werden, um möglichst rasch vom "*Programmieren im Kleinen*" (Realisierung kleiner, d.h. von einer Person in einer typischen Übungsaufgabe von 1-2 Stunden programmierbarer Algorithmen) zum "*Programmieren im Großen*" (Implementierung großer, d.h. von mehreren Personen über einen längeren Zeitraum entwickelter, komplexer Algorithmen) überzugehen.

2.1 Syntaxdarstellungen

In Kapitel 1 haben wir Grammatiken als formales Beschreibungsmittel kennengelernt, mit dem man die Syntax einer Programmiersprache definieren kann. Die Darstellung der Regeln einer Grammatik in der dort vorgestellten Form von Paaren oder als Tripel ($\langle$ linke Seite der Regel $\rangle$, $\rightarrow$, $\langle$ rechte Seite der Regel $\rangle$) ist bei Grammatiken mit sehr vielen Regeln - also auch für imperative Programmiersprachen wie MODULA-2 - oft nicht sehr übersichtlich.

Deshalb werden andere Darstellungsformen (Sprachen) gewählt. Die gebräuchlichsten, die *Syntaxdiagramme* und die *Backus-Naur-Form* (BNF) werden nachfolgend vorgestellt. Eine Sprache M zur Beschreibung einer Sprache L heißt *Metasprache*, die Sprachelemente von M enthalten *Metasymbole*, die weder Nichtterminal- noch Terminalsymbole der beschriebenen Sprache L sind. Die Notation der Syntaxdiagramme und die BNF-Notation sind in diesem Sinne Metasprachen.

Syntaxdiagramme

Die Syntax von Programmiersprachen wird vielfach, wie von N. Wirth bei PASCAL Anfang der 70er Jahre eingeführt, durch Syntaxdiagramme beschrieben.

Das Prinzip läßt sich durch eine Analogie veranschaulichen: Zwei Touristen fahren gemeinsam im Auto durch eine Region, in der es nur Einbahnstraßen gibt; sie heißt daher auch One-Way-County (siehe oberer Teil von Abb. 2.1). Der Beifahrer fotografiert alle auf der Landkarte eingezeichneten "Sehenswürdigkeiten" (durch abgerundete

Rechtecke wie RR-Station und Motel repräsentiert). Es ist nun zu überlegen, welche Bildsequenzen auf dem Film möglich sind, welche nicht.

Offenbar kann man die möglichen Bildsequenzen ermitteln, indem man die Fahrt nachvollzieht. Erreicht man dabei in der Landkarte One-Way-City (durch ein Rechteck repräsentiert), so zeigt der Stadtplan von One-Way-City (unterer Teil von Abb. 2.1) den weiteren Weg, bis man diesen wieder verläßt und am Ausgang des Rechtecks One-Way-City im oberen Teil der Abb. 2.1 in der Landkarte zu One-Way-County fortfährt.

Abb. 2.1 Landkarte von One-Way-County, Stadtplan von One-Way-City

Mögliche Bildsequenzen sind z.B. (jeweils Anfangsbuchstabe notiert): RBG, RBSMBCG. Dagegen sind beispielsweise unmöglich: RGB, RBCMG.

Ganz ähnlich werden die Syntaxdiagramme für MODULA-2 interpretiert (komplett dargestellt im Anhang B "Syntaxdiagramme für MODULA-2").

Backus-Naur-Form

Zwei ältere Notationen sind genauso mächtig wie die Syntaxdiagramme: BNF und EBNF (Extended BNF). Die BNF wurde erstmals bei der Definition der Sprache ALGOL-60 verwendet, EBNF hat sich daraus später in zahlreichen Varianten gebildet.

Das Äquivalent zu einem einzelnen Syntaxdiagramm heißt hier *Produktion*, ein Begriff, den wir in einem etwas engeren Verständnis bei der Einführung von Grammatiken kennengelernt haben.

Metasymbole in BNF sind:

$::=$	bedeutet:	ist definiert als ...
	bedeutet:	oder.

Die *Nichtterminalsymbole* (entsprechen den Rechtecken in den Syntaxdiagrammen) werden zur Unterscheidung von den Terminalsymbolen (entsprechen den Langrunden in den Syntaxdiagrammen) in spitze Klammern gesetzt, also z.B. <Programm>.

Beispiel:

Definiert wird die Bildung beliebig langer Summen aus a, b und c, also z.B. "b" oder "a+c+a". Das Startsymbol ist <Summe>.

```
<Summe>      ::= <Summand> | <Summand> + <Summe>
<Summand>    ::= a | b | c
```

EBNF und BNF sind von gleicher Ausdrucksmächtigkeit, EBNF bietet gegenüber BNF eine reichere Meta-Syntax und erfordert daher weniger, i.a. besser lesbare Produktionen:

=	"definiert als" ("::=" in BNF)
(... ...)	genau eine Alternative aus der Klammer muß stehen
[...]	Inhalt der Klammer kann stehen oder nicht
{ ... }	Inhalt der Klammer kann n-fach stehen, $n \geq 0$
.	Ende der Produktion (in BNF nicht speziell markiert).

Anders als in BNF sind hier nicht die Nichtterminal-, sondern die Terminalsymbole speziell gekennzeichnet, nämlich durch Klammerung mit Anführungszeichen ("..."). Damit wird z.B. aus dem MODULA-2-Syntaxdiagramm 48 (CaseStatement in Anhang B) in EBNF:

```
CaseStatement = "CASE" Expression "OF"
               [ CaseLabelList ":" StatementSequence ]
               {"I" [ CaseLabelList ":" StatementSequence ] }
               [ "ELSE" StatementSequence ] "END" .
```

In der Praxis setzt man die komplexeren Syntaxdiagramme meist in mehrere EBNF-Produktionen um; andernfalls werden sie durch die geschachtelte Klammerung unübersichtlich. Im obigen Beispiel entstehen so typisch folgende Produktionen:

```
CaseStatement = "CASE" Expression "OF" case {"I" case }
               [ "ELSE" StatementSequence ] "END" .
case          = [ CaseLabelList ":" StatementSequence ] .
```

Das folgende Beispiel (aus der Schweiz des Jahres 1985) zeigt, daß eine begriffliche Verwirrung des Syntaxdiagramms bereinigt werden muß, um eine Darstellung in EBNF zu erlauben (zur Erläuterung: Furgler und Egli sind Bundesräte, Egli aber auch eine Fischart). Für folgende Sätze soll eine Syntax angegeben werden:

(1) "Furgler regiert", (2) "Egli regiert", (3) "Egli schwimmt".

Mit Syntaxdiagrammen läßt sich dies wie in Variante (a) der nächsten Abbildung ausdrücken. Die direkte Umsetzung nach EBNF ist aber erst aus der Variante (b) möglich, die auch semantisch klarer ist.

Abb. 2.2 Nicht BNF-konformes und BNF-konformes Syntaxdiagramm

Die EBNF-Form ist damit:

Satz = PolitSatz | ZooSatz .
 PolitSatz = ("Furgler" | "Egli") "regiert".
 ZooSatz = "Egli" "schwimmt".

Bezüglich der Ausdrucksmächtigkeit besteht kein prinzipieller Unterschied zwischen Syntaxdiagrammen und EBNF. In der Praxis gibt es aber gute Gründe, sich für das eine oder andere zu entscheiden: Für die Syntaxdiagramme spricht die hohe *Anschaulichkeit* und die *geringe* Zahl der Regeln (d.h. Diagramme). Eine EBNF-Grammatik läßt sich wesentlich leichter erstellen und ändern, denn sie kann mit jedem alphanumerischen Gerät bearbeitet werden. Sie erzwingt eine *klare* Struktur und kann u.U. von einem Übersetzer-erzeugenden System (sog. Compiler-Compiler) *direkt* verwendet werden.

Damit sind Syntaxdiagramme immer dann vorzuziehen, wenn sich der hohe Aufwand lohnt (z.B. für ein Lehrbuch). Für wenige, versierte Leser sind dagegen EBNF-Grammatiken ausreichend (z.B. für einen Sprachentwurf in einer Diplomarbeit). Die einfache BNF hat nur noch geschichtliche Bedeutung.

Die Abwägung wird hinfällig, wenn man über ein System verfügt, das EBNF automatisch in Syntaxdiagramme wandelt. In diesem Falle ist nur noch zu berücksichtigen, daß in einem Syntaxdiagramm typisch mehrere Produktionen der EBNF zusammengefaßt sind.

2.2 Elementare funktionale Modula-2-Programme

Wir werden uns bei der Einführung von MODULA-2 zunächst auf Sprachkonzepte beschränken, die zur Realisierung einfacher *funktionaler* Programme genügen. Dies sind Programme, die ohne Speicherung von Zwischenergebnissen und damit *ohne Variablen* auskommen.

Dieser Ansatz wird alle überraschen, die bereits Programmiererfahrung haben. Warum wird ein so wichtiges Konzept wie das der Variablen zunächst zurückgehalten? Wir haben dafür zwei Gründe.

- Aus didaktischer Sicht wollen wir deutlich machen, daß es hier primär nicht um einen Programmierkurs, sondern um das Erlernen der Konzepte geht. Die Erfahrung zeigt, daß gerade diejenigen, die bei einer schrittweisen Einführung in scheinbar längst Vertrautes nur müde abwinken, die neuen Konzepte nicht bemerken und später größte Mühe haben, den Anschluß zu finden. Ihnen wollen wir mit dem Neuen ein Signal geben.
- Aus Sicht der Informatik ist die Variable keine Naturgegebenheit, sondern ein durch die Von-Neumann-Architektur eingeführtes, äußerst praktisches *Realisierungskonzept*. Daß dieses seine Tücken hat, wird im Abschnitt über die Verifikation sichtbar werden. Neuere Ansätze (z.B. funktionale und logische Programmierung) zeigen dann, daß durchaus auch andere, "höhere" Konzepte möglich sind, bei denen die Variablen nur noch bei der Übersetzung entstehen, gerade wie die Sprunganweisung (GOTO), die zunächst ebenfalls unentbehrlich schien, in den modernen Programmiersprachen aber nicht mehr angeboten wird. Schon 1973 hat F.L. Bauer den berühmten Aufsatz von Dijkstra "GOTO considered harmful" zitiert, indem er "GOTO" durch "Variables" ersetzte. Der vorläufige Verzicht auf die Variablen ist also ein Votum für einen Ansatz, der die *Einfachheit* vor die Effizienz stellt.

2.2.1 Eine MODULA-2-Teilsprache

Ein *Bezeichner* (identifizier) in MODULA-2 besteht aus Buchstaben und Ziffern, wobei am Anfang ein Buchstabe stehen muß (siehe Abb. 2.3; die vollständige Definition der Buchstaben ist hier aus Raumgründen nur angedeutet, dicke Linien werden später erläutert). Groß- und Kleinbuchstaben werden unterschieden, so daß z.B. "Alpha" ungleich "alpha" ist.

Die kleinen Indizes haben in der Syntax keine Bedeutung, sie dienen nur der Orientierung; dieser Sinn wird allerdings nur mit dem Anhang B der vollständigen MO-

DULA-2-Syntax deutlich. Wo der Index durch “/” modifiziert ist, ist die Definition gegenüber dem “vollen” MODULA-2 noch eingeschränkt.

Abb. 2.3 Syntaxdiagramme zur Definition der Bezeichner

Ähnlich wie die Bezeichner werden nun *Strings* (Zeichenketten) definiert. Die Auswahl der Sonderzeichen ist maschinenabhängig (deshalb kursiv in Abb. 2.4), beispielsweise können die Umlaute dazugehören oder nicht.

Abb. 2.4 Syntax von Zeichen und Strings

Wir führen nun Sprachkonstrukte ein, die wir generell zur Beschreibung des “*Programmrahmens*”, d.h. der Struktur von MODULA-2-Programmen bzw. Programmteilen (*Module* genannt) und damit auch schon für die MODULA-2-Teilsprache benötigen.

Ein *MODULA-2-Programm* besteht aus einem Hauptmodul, dem *ProgramModule*, sowie evtl. weiteren Modulen. Anschaulich kann man sich MODULA-2-Programme wie folgt abgestuft vorstellen:

- *Module* als große, möglicherweise getrennt entwickelte (Teil-)Programme,
- (*Funktions-*)*Prozeduren* als kleinere, aufrufbare funktionale Einheiten in einem Modul und
- *Blöcke* als logisch zusammenhängende Anweisungsfolgen.

Durch eine entsprechende Definition besitzen Module eigene, in ihnen festgelegte Prozeduren, können aber auch Prozeduren anderer Module nutzen (“*importieren*”). Dies muß in der Definition des Modulrahmens ebenfalls explizit festgelegt werden (vgl. Abb. 2.5).

- *ProgramModule*: der “Rahmen” eines jeden Programms bzw. Moduls. Er enthält einen (ihn eindeutig identifizierenden) Namen, eine Importliste mit den aus anderen Modulen verwendeten (d.h. importierten) Prozeduren (s.u.) und - in einem sogenannten Block (s.u.) zusammengefaßt - die Anweisungen des Moduls;
- *Import*: die Festlegung, aus welchem Modul welche Prozeduren verfügbar sein sollen;

- **Block:** Folge von *Anweisungen* (Statements), eingeschlossen in die Terminalsymbole BEGIN und END;
- **ProcedureCall:** Aufruf einer Prozedur, d.h. eines Blocks, der unter einem eindeutigen Namen an einer anderen Stelle im gleichen oder in einem anderen Modul notiert ist.

Abb. 2.5 Syntaxdiagramme für Modulrahmen

Man beachte, daß Syntaxdiagramme - wie auch die Notationen BNF und EBNF - den Syntax-Begriff aus dem Kapitel 1, dem typisch die Prüfungen eines Compilers entsprechen, nicht abdecken, denn sie beschreiben nur die kontextfreie Syntax;

kontextsensitive Bedingungen können mit Syntaxdiagrammen nicht dargestellt werden. Daher ist das Programm

```
MODULE X; FROM InOut IMPORT (* PROC *) WriteLn; BEGIN END Y.
```

im Sinne der kontextfreien Syntax korrekt; die Bezeichner am Anfang und Ende (X und Y) sind aber verschieden, so daß das Programm von keinem MODULA-2-Compiler akzeptiert werden kann. Die kontextfreie Syntax muß also noch durch *Kontextbedingungen* (auch Attribute genannt) ergänzt werden.

Beispiele solcher Kontextbedingungen für die oben vorgestellten Elemente sind z.B. für

- ProgramModule: die beiden Bezeichner am Anfang und am Ende müssen gleich sein;
- Import: vorerst kommt für den ersten Bezeichner nur InOut in Frage; in der Liste der Prozeduren können WriteString und WriteLn stehen;
- ProcedureCall: als Bezeichner kommen nur die Namen aus der obigen Import-Liste in Frage.

Die Syntax ist nicht exakt bezüglich der *Leerzeichen* (Blanks). Hier gilt folgende Regel: Blanks können an beliebiger Stelle eingestreut werden außer dort, wo in den Syntaxdiagrammen dicke Linien stehen. Ein Blank ist dort *notwendig*, wo sonst die syntaktische Struktur nicht erkennbar wäre, also beispielsweise bei einem leeren Block zwischen BEGIN und END. Kommentar und Zeilenende werden wie ein Blank behandelt. Die Strukturierung in Wörter, Zahlen und Symbole (z.B. “;”, “:=”) wird - wie in Kapitel 1 ausgeführt - bei einem Übersetzer durch eine relativ einfache Komponente, den Scanner, vorgenommen. Dieser braucht die Trennzeichen, um die Wortgrenzen zu erkennen.

Ein erstes Programm-Beispiel mit der bisher vorgestellten Syntax ist das folgende Programm P2.1 (die äußere Gestalt, das *Layout*, dieses Programms ist noch unbefriedigend und wird später verbessert):

```
MODULE Ueberschrift; (* Programm P2.1 *)
FROM InOut IMPORT (* PROC *) WriteString, WriteLn; BEGIN
WriteString('Informatik I'); WriteLn; WriteString (
'Wintersemester 1989/90'); WriteLn END Ueberschrift.
```

Andere Einschränkungen als die oben genannten Kontextbedingungen könnten zwar prinzipiell durch Syntaxdiagramme angegeben werden, doch würde dies zu kompliziert. So dürfen Wortsymbole nicht als Bezeichner verwendet werden, obwohl sie nach den Syntaxdiagrammen erlaubt wären. Ihr Ausschluß wäre außerordentlich kompliziert. Als Beispiel diene der Ausschluß des Wortsymbols “OF” mittels Syntaxdiagramm in Abb. 2.6.

Abb. 2.6 Syntaxdiagramme für den Ausschluß des Wortes "OF"
(Die Produktionen für LetterExceptO und Letter ExceptF fehlen)

Der Aufwand, Ausnahmen mittels Diagrammen zu beschreiben, wächst sehr rasch mit der Komplexität der Regeln. Die Möglichkeit, alle Ausnahmen formal zu erfassen wie beispielsweise das Verbot, Wortsymbole als Bezeichner zu verwenden, besteht also nur theoretisch. Darum beschreibt man solche Verbote durch textuelle Zusätze, womit natürlich der Boden kontextfreier Beschreibungen verlassen wird!

Syntaxdiagramme können auch *rekursiv* sein, wir werden später davon Gebrauch machen. Auf den weiter oben benutzten Vergleich mit Landkarten läßt sich das schwer übertragen; es ist vielleicht vergleichbar mit einer Situation, in der man in einem Gebäude das Stockwerk wechselt und wieder in ein strukturell völlig gleiches gerät (typisch für Parkhäuser).

Abb. 2.7 zeigt, wie eine beliebig tief geklammerte Ziffer beschrieben wird, Beispiele für die aus Klammer-Ziffer erzeugten Zeichenreihen sind "7", "(((4)))" und "(((((((0))))))))."

Abb. 2.7 Beispiel eines rekursiven Syntaxdiagramms

Im Vergleich zu den noch in Kapitel 4 vorzustellenden Ansätzen einer formalen *Semantik*-Beschreibung begnügen wir uns an dieser Stelle für die MODULA-2-Teilsprache mit einer informellen, verbalen Beschreibung ihrer Bedeutung.

ProgramModule: MODULE ABC; ... END ABC.

Das Objekt ABC kann - nach den notwendigen Vorbereitungen, insbesondere der Übersetzung - dem Laufzeitsystem des Rechners zur Ausführung übergeben werden.

Import: FROM M IMPORT A, ... , Z;

Die in der Liste genannten Objekte A bis Z (hier zunächst nur Prozeduren) aus dem Modul M stehen im Block zur Verfügung.

Block und StatementSequence: BEGIN S₁; ... ; S_n END

Die Anweisungsfolge S₁ bis S_n im Block wird bei Ausführung des Program-Module in der Reihenfolge der Aufschreibung abgearbeitet.

Statement (leere Anweisung, ProcedureCall): P(E₁, ..., E_n) oder P

Die leere Anweisung hat keine Wirkung.

Der Prozeduraufruf bewirkt die Ausführung der Prozedur, ggf. mit den in der Klammer stehenden Parametern. Vorerst stehen nur folgende Prozeduren zur Verfügung:

- WriteString("Eine Zeichenreihe"): der String (hier also "Eine Zeichenreihe") wird in die laufende Zeile der Ausgabe übertragen
- WriteLn: die laufende Zeile der Ausgabe wird abgeschlossen.

Bemerkungen zum Programmierstil

Man sollte sinnvolle, "sprechende" *Bezeichner* für Programme, Prozeduren usw. wählen, keine kryptischen Kürzel wie "MTNNY" oder nur einzelne Buchstaben. Das Programm sollte nicht nur logisch, sondern auch optisch klar gegliedert sein. Ebenso sollten Wortsymbole wie z.B. BEGIN und END durch *Kommentare* ergänzt werden, um das Programm verständlicher zu machen.

Kommentare werden in Kommentar-Klammern (* ... *) eingeschlossen und dürfen geschachtelt werden, z.B. (* Dies (* ist (* ein *) geschachtelter *) Kommentar *).

Diese Schachtelung ist aber nur sinnvoll, um ganze Programmteile, die auch Kommentare enthalten, zu unterdrücken, ohne sie zu löschen (beispielsweise für Testzwecke).

Das folgende Programm P2.2 stellt eine kommentierte Version des Programms P2.1 dar und ist ein Beispiel für eine ausführliche und übersichtliche Erläuterung der Funktion eines Moduls. In den weiteren Programmen wird aus Platzgründen auf diese Kommentarblöcke weitgehend verzichtet, weil die Angaben im Rahmen eines Lehrbuchs ohne Bedeutung wären; Studenten sollten sich aber von Beginn an daran gewöhnen.


```

MODULE Ueberschrift;                                (* Programm P2.2 *)
(* Ausgabe zweier Titelzeilen f. die Vorlesung Informatik I *)
(* $PROJEKT: Einführung in die MODULA-2-Programmierung *)
(* $DATEI: Prog2.mod $VERZEICHNIS: /usr/local/src/MODULA-2 *)
(* $ERSTELLT: 1985-11-20 $LETZTE AENDERUNG:1989- 4-12 *)
(* $AUTOR: J. Ludewig, ETH Zuerich $VERSION: 1.0 *)
(* $BEMERKUNG: Demonstration des Layouts und der Kommentare *)
FROM InOut IMPORT (* PROC *) WriteString, WriteLn;
BEGIN (* Ueberschrift *)
  WriteString ('Informatik I');           WriteLn;
  WriteString ('Wintersemester 1989/90'); WriteLn;
END Ueberschrift.

```

Zahlen, symbolische Konstanten und Ausdrücke

Nachfolgend erweitern wir die MODULA-2-Teilsprache, indem wir Zahlen, Ausdrücke, Ausgabe-Anweisungen und symbolische Konstanten einführen.

Ganze Zahlen werden in MODULA-2 durch eine Folge von Ziffern dargestellt und als INTEGER-Zahlen bezeichnet. Zahlen mit Punkt und eventuell Exponenten dienen zur (allgemein nur näherungsweise möglichen) Darstellung *reeller Zahlen* man nennt sie daher REAL-Zahlen. Es sollte aber bei ihrem Gebrauch völlig klar sein, daß reelle Zahlen im Sinne der Mathematik auf dem Rechner wegen der endlichen Darstellung und der internen Kodierung als Dualzahlen nicht zur Verfügung stehen.

Ausdrücke können durch Anwendung arithmetischer und logischer Operatoren auf Zahlen, Strings oder Ausdrücke (rekursive Definition) gebildet werden. Man unterscheidet dabei *arithmetische* und *logische* Ausdrücke.

Zur Erweiterung der MODULA-2-Teilsprache werden diese neuen Sprachkonstrukte in Abb. 2.8 anhand ihrer Syntaxdiagramme vorgestellt.

Expression₃₉

SimpleExpression_{40/*}

Relation $_{28}r$

"<>" und "#" haben beide die Bedeutung "ungleich".

MulOperator $_{31}r$ Integer $_{4}r$

Abb. 2.8 Zusätzliche Syntaxdiagramme

Wir erlauben nun, neben Strings auch Zahlen und arithmetische Ausdrücke auszugeben. Dazu wird aus dem Modul InOut zusätzlich WriteInt importiert.

Aufrufe dieser Prozedur haben die Form WriteInt (K1, K2) und folgende Bedeutung: Die Zahl oder der Ausdruck K1 wird mit K2 Stellen ausgegeben. Dabei wird Null durch eine Ziffer null dargestellt; im übrigen werden führende Nullen unterdrückt. Ein Vorzeichen erscheint nur bei negativen Zahlen. Steht mehr Platz als nötig zur Verfügung, so werden vor (also links von) der Zahl entsprechend viele Blanks ausgegeben (die Ausgabe erfolgt also *rechtsbündig*). Reicht der Platz nicht aus, so ist das Verhalten undefiniert; die meisten Systeme ignorieren dann K2 und geben die Zahl ganz aus.

Hier ist zu beachten, daß bei der Definition von MODULA-2 keine “*Standard-Bibliothek*”, d.h. eine Sammlung von Basis-Modulen mit standardisierten Prozeduren etwa zur Ein-Ausgabe (E/A) festgelegt wurde; die Sprachdefinition von Wirth gibt diesbezüglich nur eine Empfehlung. Darum ist die Semantik z.B. der E/A-Befehle leider stark systemabhängig, was die Portabilität, also die Übertragbarkeit eines Programms auf eine andere Maschine, beeinträchtigt.

Für die Ausgabe von REAL-Zahlen muß WriteReal importiert werden. Diese Prozedur gehört aber zum Modul RealInOut, so daß eine separate Import-Anweisung erforderlich ist. Aufrufe dieser Prozedur haben die Form: WriteReal(K1,K2).

Die Bedeutung der Parameter K1 und K2 ist analog zur Prozedur WriteInt. Die Ausgabe erscheint in normierter Exponential-Darstellung, z.B. in der Form “-0.31415E+01” für -3.13415. Darum muß K1 mindestens den Wert 8 besitzen.

Für die Ausgabe von Zahlen wird die Definition von ProcedureCall wie in Abb. 2.9 dargestellt erweitert.

Abb. 2.9 Syntaxdiagramm für Prozeduraufruf

Beispiel eines Programms:

```

MODULE Ballonfahrer;                                (* Programm P2.3 *)
FROM InOut      IMPORT (* PROC *) WriteString, WriteLn;
FROM RealInOut  IMPORT (* PROC *) WriteReal;
BEGIN (* Ballonfahrer *)
  WriteString ('Gesamtgewicht = ');
  WriteReal (78.3 + 77.1 + 61.9, 8);
  WriteString (' kg'); WriteLn;
END Ballonfahrer.
  
```

Die Verwendung von *Konstanten* ("Literalen") in Ausdrücken (z.B. 78.3 und 'kg') führt zu Problemen beim Lesen und beim Korrigieren. Darum werden neue *symbolische Konstanten* (oder oft einfach: *Konstanten*) eingeführt. Sie werden nach dem Wortsymbol CONST vor dem Block definiert:

```

CONST ZahlStudenten = 199;
      Pi = 3.1415;
      Einheit = 'kg';
  
```

Die Syntax ist in Abb. 2.10 definiert. Block (Syntaxdiagramm 12) ersetzt nun die alte Fassung ohne Deklarationen.

Block 12

ConstDeclaration₂₆

Abb. 2.10 Syntaxdiagramm für Konstanten-Definition

ConstExpression (Syntaxdiagramm 27) ist ganz analog wie Expression definiert (Syntaxdiagramm 39), wobei darin aber nur Literale und bereits definierte Konstanten vorkommen dürfen.

```
CONST FaktorUmfang = 2.0 * Pi;
```

Das Ballonfahrer-Programm P2.3 sieht nun wie folgt aus:

```

MODULE Ballonfahrer;                                     (* Programm P2.4 *)
(* Variante mit symbolischen Konstanten *)
FROM InOut      IMPORT (* PROC *) WriteString, WriteLn;
FROM RealInOut  IMPORT (* PROC *) WriteReal;
CONST GewichtAlfred = 78.3; (* kg *)
      GewichtBertha  = 77.1; (* kg *)
      GewichtClaudia = 61.9; (* kg *)
      LngReal        = 10;
      Groesse = 'Gesamtgewicht = ';
      Einheit = 'kg';

BEGIN (* Ballonfahrer *)
  WriteString (Groesse);
  WriteReal (GewichtAlfred + GewichtBertha + GewichtClaudia,
            LngReal);
  WriteString (Einheit); WriteLn;
END Ballonfahrer.

```

2.2.2 Programmverzweigungen

Durch die nun folgenden Erweiterungen der bisherigen MODULA-2-Teilsprache werden Programmverzweigungen möglich.

Programmverzweigungen erlauben, abhängig vom aktuellen *Programmzustand* unterschiedliche Anweisungen auszuführen. Wir haben dazu zunächst die Konstruktion der *Fallunterscheidung* (Case-Statement). Die Syntax ist in Abb. 2.11 angegeben. Nicht angegeben ist die Erweiterung der Definition von Statement (45/1), durch die auch das CaseStatement alternativ zur leeren Anweisung und zum Prozeduraufruf zugelassen wird.

Die Anordnung der verschiedenen Fälle im Case-Statement spielt keine Rolle, jeder Fall darf aber höchstens einmal vorkommen. Ist der Wert des Ausdrucks hinter CASE

mit keiner *Fall-Liste* (CaseLabelList) abgedeckt, so werden die auf ELSE folgenden Anweisungen ausgeführt; der ELSE-Teil muß in diesem Fall vorhanden sein.

CaseStatement₄₈ CaseLabelList₂₄

Abb. 2.11 Syntax der CASE-Anweisung

Die Anwendung der Verzweigung soll an Programm P2.5 gezeigt werden, das zur Nummer eines Tages den Wochentag ausgibt.

```

MODULE Wochentag;                                     (* Programm P2.5 *)
(* Ausgabe des Wochentages zur Nummer eines Tages im *)
(* Monat zur Demonstration der Verzweigung *)
FROM InOut IMPORT (* PROC *) WriteString, WriteLn, WriteInt;
CONST NrMo=1; NrDi=2; NrMi=3; NrDo=4; NrFr=5; NrSa=6; NrSo=7;
  WochenLaenge = NrSo;
  ersterMontag = 3;
  Heute = 7;      (* laufender Monat, Tag heute *)
BEGIN (* Wochentag *)
  WriteString ('Heute, am '); WriteInt (Heute, 3);
  WriteString ('., ist ');
  CASE (Heute - ersterMontag + WochenLaenge) MOD
    WochenLaenge + 1 OF
  | NrMo: WriteString ('Montag')

```



```

| NrDi: WriteString ('Dienstag')
| NrMi: WriteString ('Mittwoch')
| NrDo: WriteString ('Donnerstag')
| NrFr: WriteString ('Freitag')
| NrSa: WriteString ('Samstag')
| NrSo: WriteString ('Sonntag')
END (* CASE *);
WriteLn;
END Wochentag.

```

Bemerkungen:

1. Die Addition von WochenLaenge vor Anwendung der MOD-Operation ist notwendig, da MOD für negative Argumente nicht definiert ist.
2. Der Trennstrich vor NrMo ist natürlich überflüssig, erleichtert aber - ähnlich einem Semikolon vor END - Ergänzungen des Codes.
3. Wir können auch die Arbeitstage gleich behandeln, indem wir eine Liste bilden (Programmvariante P2.5a):

```

CASE (Heute - ersterMontag + WochenLaenge) MOD
WochenLaenge + 1 OF
| NrMo, NrDi, NrMi, NrDo, NrFr: WriteString ('Arbeitstag')
| NrSa: WriteString ('Samstag')
| NrSo: WriteString ('Sonntag')
END (* CASE *);

```

4. Die Liste von Montag bis Freitag kann, da sie einen zusammenhängenden Wertebereich darstellt, abgekürzt werden zu NrMo..NrFr (Programmvariante P2.5b):

```

CASE (Heute - ersterMontag + WochenLaenge) MOD
WochenLaenge + 1 OF
| NrMo..NrFr: WriteString ('Arbeitstag')
| NrSa: WriteString ('Samstag')
| NrSo: WriteString ('Sonntag')
END (* CASE *);

```

5. Oder die Arbeitstage werden mit dem ELSE-Zweig behandelt (Programmvariante P2.5c):

```

CASE (Heute - ersterMontag + WochenLaenge) MOD
WochenLaenge + 1 OF
| NrSa: WriteString ('Samstag')
| NrSo: WriteString ('Sonntag')
ELSE WriteString ('Arbeitstag')
END (* CASE *);

```

Für den Spezialfall der "Entweder-Oder"-Entscheidung in Abhängigkeit von einem *booleschen Ausdruck* (der nur die Werte TRUE und FALSE annehmen kann) gibt es in MODULA-2 das Konstrukt IF...THEN...ELSE...END mit der in Abb. 2.12 definierten Syntax.

Abb. 2.12 Syntax der IF-Anweisung

Wenn im Programm P2.5 Wochentag nur zwischen Werktag und Sonntag unterschieden werden soll, genügt die einfache Verzweigung:

```

MODULE Wochentag;                                     (* Programm P2.6 *)
(* Variante zur Unterscheidung Werktag / Sonntag *)
(*... ( Programm wie in P2.5 ) *)
  IF (Heute - ersterMontag + WochenLaenge) MOD WochenLaenge + 1
                                         = NrSo THEN
    WriteString ('Sonntag');
  ELSE
    WriteString ('Werktag');
  END (* IF *);
  WriteLn;
END Wochentag.

```

Läßt man den ELSE-Zweig in einer IF-Anweisung weg, so ergibt sich eine *bedingte Anweisung*, deren Inhalt nur ausgeführt wird, falls die entsprechende Bedingung erfüllt (d.h. TRUE) ist; andernfalls wird nach dieser Anweisung im Programm fortgefahren. Im Gegensatz zur CASE-Anweisung müssen hier also nicht alle Fälle abgedeckt sein.

Ein Beispiel ergibt sich in Fortführung des Programms P2.4, wenn wir ausdrucken, ob der Ballon abheben kann:

```

MODULE Ballonfahrer;                                  (* Programm P2.7 *)
(* Variante mit bedingter Anweisung *)
FROM InOut      IMPORT (* PROC *) WriteString, WriteLn;
FROM RealInOut  IMPORT (* PROC *) WriteReal;

```



```

CONST GewAlfred   = 78.3; (* kg *)
      GewBertha   = 77.1; (* kg *)
      GewClaudia  = 61.9; (* kg *)
      MaxNutzlast = 210.0; (* kg *)
      LngReal     = 10;
BEGIN (* Ballonfahrer *)
  WriteString ('Beladung = ');
  WriteReal  (GewAlfred + GewBertha + GewClaudia, LngReal);
  WriteString (' kg'); WriteLn;
  IF GewAlfred + GewBertha + GewClaudia > MaxNutzlast THEN
    WriteString ('Start ist nicht moeglich. Uebergewicht = ');
    WriteReal  (GewAlfred + GewBertha + GewClaudia -
                MaxNutzlast, LngReal);
  END (* IF *);
  WriteLn;
END Ballonfahrer.

```

Die mehrfache Aufschreibung der Summe ist nicht sehr elegant; dieses Problem wird später durch Funktionen oder Variablen gelöst. Durch das letzte Semikolon entstehen im Beispiel an zwei Stellen *leere Anweisungen*. Ihr Vorteil ist, daß die Syntax ohne weiteres korrekt bleibt, wenn am Ende noch ein Befehl zugefügt wird.

Die bisher vorgestellte MODULA-2-Teilsprache ermöglicht also Programmverzweigungen durch

- *Mehrfache Alternative:* CASE...END
- *Einfache Alternative:* IF...THEN...ELSE...END
- *Bedingte Anweisung:* IF...THEN...END

2.2.3 Funktionen und Prozeduren

Wir erweitern nun die MODULA-2-Teilsprache um Möglichkeiten, die die Programmierung von Funktionen, Prozeduren und Programmeingaben erlaubt. Auf entsprechende Syntaxdiagramme verzichten wir im folgenden. Sie können dem Anhang B "Syntaxdiagramme für MODULA-2" entnommen werden.

Funktionen

Der Funktionsbegriff stammt aus der Mathematik; er bezeichnet eine *Abbildung*, beispielsweise die Abbildung $y = f(x) = x^2$. Funktionen sind nicht auf Zahlen beschränkt, wir können ebensogut Abbildungen zwischen Buchstaben oder anderen Objekten zulassen.

Bei der Vorstellung von Grundelementen imperativer Programmiersprachen hat der Funktionsbegriff eine etwas andere Bedeutung, wie Abb. 2.13 in der Gegenüberstellung der Verwendung der Wörter "Funktion" und "Parameter" in der Mathematik und in der imperativen Programmierung zeigt.

	a (Element aus A)	b=f(a) (Element aus B)	A	B
Mathematik	unabhängig Veränderliche, Argument	abhängig Veränderliche, Funktion	Defin.- bereich	Bild- bereich
Imperative Program- mierung	Parameter algebraisch: formaler Param. spezieller Wert: konkreter Param.	Funktionswert (allgemein oder für konkrete, bzw. "aktuelle" Parameter)	Typ d. Param. a Datentypen	Typ d. Funkt. f(a)

Abb. 2.13 Der Funktionsbegriff in Mathematik und imperativer Programmierung

Allgemein ist zu bemerken, daß die mathematische Funktion einen *statischen* Zusammenhang beschreibt, die der imperativen Programmierung dagegen eine *dynamische* Berechnungsanweisung. Diese Gegenüberstellung gilt im übrigen nicht mehr in bestimmten "nicht-imperativen" Programmiersprachen, die wir später kennenlernen werden. Zur Definition einer Funktion gehören folgende Angaben:

- das Terminalsymbol **PROCEDURE**;
- der *Bezeichner*, der die Funktion repräsentiert;
- der zugeordnete *Typ* (Bildbereich);
- für jeden *Parameter* ein Bezeichner und der Typ;
- der formale Zusammenhang zwischen Parametern und Funktionswert (*Berechnungsvorschrift*).

Beispiel:

Eine Funktion soll die Masse eines Ziegelsteins mit bestimmten Maßen liefern. Mit einer Dichte von $1,5 \text{ g/cm}^3$ ergibt sich (bei Angabe der Maße in cm und der Masse in kg) die folgende Funktion:

Bezeichner: Ziegelmasse Typ: REAL;
 Parameter 1: Laenge Typ: REAL;
 Parameter 2: Breite Typ: REAL;
 Parameter 3: Hoehe Typ: REAL;
 Zusammenhang: Ziegelmasse = Laenge * Breite * Hoehe * Dichte.

Eine Realisierung in MODULA-2 sieht z.B. so aus:

```

(* Programm P2.8 *)
PROCEDURE Ziegelmasse (Laenge, Breite, Hoehe: REAL): REAL;
(* Ziegelmasse: Berechnung der Masse eines Ziegels *)
(* $PARAMETER: Laenge /IN/ Laenge eines Ziegels *)
(*           Breite /IN/ Breite eines Ziegels *)
(*           Hoehe /IN/ Hoehe eines Ziegels *)

```

```

(* $RESULTAT: Ziegelmasse in kg *)
(* $VORBEDINGUNG: Laenge > 0, Breite > 0, Hoehe > 0 *)
(* $FUNKTION: Berechnung der Masse eines Quaders (in *)
(* kg) mit den als Parametern gegebenen Abmessungen *)
(* (in cm). Die Dichte ist konstant. *)
(* $BEMERKUNG: ... *)
CONST Dichte = 1.5E-3; (* in kg/ccm *)
BEGIN (* Ziegelmasse *)
    RETURN Laenge * Breite * Hoehe * Dichte;
END Ziegelmasse;

```

Prozeduren

Prozeduren, die wir bisher schon für die Ein-/Ausgabe verwendet haben, sind den Funktionen sehr ähnlich. Sie werden (ebenfalls mit dem Wortsymbol PROCEDURE) vor dem eigentlichen Programmblock definiert (zur Syntax siehe Diagramm 56 in Anhang B). Der wesentliche Unterschied zu Funktionen besteht darin, daß Prozeduren kein Ergebnis liefern. Dies hat die folgenden Konsequenzen:

- Der Aufruf einer Prozedur stellt keinen Ausdruck (Expression) dar, sondern eine *Anweisung*.
- Die Definition der Prozedur enthält nach der Liste der formalen Parameter *keinen* Doppelpunkt und Ergebnistyp.

Anders als eine parameterlose Funktion wird eine solche Prozedur ganz ohne Klammer aufgerufen. Ein Beispiel folgt später mit dem Spiel "Türme von Hanoi". Da eine Prozedur kein Ergebnis liefert, kann ihr Sinn nur in einer *Nebenwirkung* (z.B. der Ausgabe von Meldungen) liegen.

2.2.4 Elementare Datentypen, Aufzählungs- und Bereichstypen

Elementare Datentypen

Wie in Programmiersprachen üblich ist auch bei MODULA-2 eine Reihe von einfachen, sogenannten elementaren Datentypen bereits *vordefiniert*, von denen wir INTEGER, CARDINAL, REAL und CHAR vorstellen werden.

Der Datentyp *INTEGER* ist eine echte (weil endliche) Teilmenge der *ganzen Zahlen* er *vertritt* in MODULA-2 die ganzen Zahlen. Die Prozeduren MIN (...) und MAX (...) sind für alle elementaren Typen und Aufzählungstypen (s.u.) vordefiniert. MIN (INTEGER) und MAX (INTEGER) geben die *Grenzen* des Integer-Bereichs an. Somit gilt für jede beliebige Integer-Zahl x :

x ist eine ganze Zahl, $\text{MIN (INTEGER)} \leq x \leq \text{MAX (INTEGER)}$.

Das folgende simple Beispiel zeigt, daß man den Unterschied zwischen INTEGER und ganzer Zahl nicht vergessen darf:

```
(* Programm P2.9 *)
PROCEDURE Doppelt (ArgDoppelt: INTEGER) : INTEGER;
BEGIN (* Doppelt *)
  RETURN 2 * ArgDoppelt;
END Doppelt;
```

Hier handelt es sich nur um eine *partielle* Funktion, weil ArgDoppelt nicht jeden Wert aus seinem Wertebereich annehmen darf; ist ArgDoppelt größer als $\text{MAX}(\text{INTEGER})/2$ oder kleiner als $\text{MIN}(\text{INTEGER})/2$, so kommt es bei der Multiplikation zum *Zahlenüberlauf*.

Für die *natürlichen Zahlen* (mit null) steht mit *CARDINAL* ein anderer Datentyp zur Verfügung. Sein Vorteil ist der größere positive Bereich bei gleichem Speicherbedarf. Wir verwenden ihn nur dann, wenn er in Zusammenhang mit vorgegebenen Prozeduren verlangt ist, oder wenn wir sicher sein können, nicht in den Bereich der negativen Zahlen zu gelangen. Durch den Übergang zu Rechnern größerer Wortlänge (z.B. erlauben 32 bit schon einen INTEGER-Bereich von -2^{31} bis $+(2^{31} - 1)$ darzustellen) gibt es für *CARDINAL* heute eigentlich keinen Bedarf mehr.

REAL ist eine endliche Menge *rationaler Zahlen*, die im Rechner die rationalen und reellen Zahlen der Mathematik vertreten. Der Wertebereich ist also endlich, und die *Genauigkeit* der Darstellung einer rationalen oder reellen Zahl ist begrenzt. Typisch sind im Rechner Zahlen der Form $\pm n * 2^{\pm m}$ exakt darstellbar, wobei m und n Integer-Zahlen sind. Dagegen sind nicht nur die transzendenten und die übrigen nicht-rationalen Zahlen (z.B. e , $\sqrt{2}$) ausgeschlossen, sondern auch Brüche wie $1/3$ und sogar Dezimalbrüche wie 0.1 .

Außerdem ist die Rechner-Arithmetik mit *REAL*-Zahlen *Rundungsfehlern* unterworfen und gehorcht daher auch im Bereich der darstellbaren Zahlen nicht den Gesetzen der Mathematik. Z.B. ist i.a. das Ergebnis eines Rechners für $A + B - B$, wobei A

und B REAL-Zahlen sind und A wesentlich kleiner als B ist, ungleich A, denn bei der Addition können nicht alle Stellen von A berücksichtigt werden.

Literal-Konstanten des Typs REAL werden als Dezimalbruch (mit Dezimal-Punkt entsprechend der in Amerika üblichen Form) geschrieben; ein durch "E" abgetrennter *Exponent* kann folgen.

Ein Ausdruck des Typs *BOOLEAN* kann einen Wert aus der Menge {FALSE, TRUE} annehmen. Damit ist BOOLEAN ein Aufzählungstyp (s. nächster Abschnitt). FALSE und TRUE bilden ein Alphabet.

Beispiel:

```
(* Programm P2.10 *)
PROCEDURE Gerade (Arg : INTEGER) : BOOLEAN;
BEGIN (* Gerade *)
  RETURN Arg MOD 2 = 0;
END Gerade;
```

Diese Funktion kann anstelle eines booleschen Ausdrucks verwendet werden, z.B. durch "IF Gerade (x+y+z) THEN ...". Dabei sind Arg formaler Parameter und x+y+z ein konkreter (auch aktueller) Parameter.

CHAR ist die Menge der verfügbaren Zeichen (systemabhängig!), d.h. der Buchstaben, Ziffern und Sonderzeichen; dazu gehört auch das Leerzeichen (Blank). Konstanten dieses Typs werden durch das Zeichen in Hochkommata oder Anführungszeichen dargestellt.

Beispiel:

```
(* Programm P2.11 *)
PROCEDURE IstZiffer (Zeichen : CHAR) : BOOLEAN;
(* keine sonderlich elegante Lösung *)
BEGIN (* IstZiffer *)
  RETURN (Zeichen = '0')
    OR (Zeichen = '1')
    (* ... usw. bis *)
    OR (Zeichen = '9');
END IstZiffer;
```

Vollständig definiert ist jeder der hier vorgestellten (Daten-)Typen durch seine *Wertemenge* und die zulässigen *Operationen*.

Beispiel: INTEGER

Wertebereich: ganze Zahlen von MIN (INTEGER) bis MAX (INTEGER)

Operationen (unvollständig):

monadisch:	+, -	INTEGER → INTEGER
dyadisch:	+, -, *, DIV, MOD	(INTEGER, INTEGER) → INTEGER
	=, <, <=, >, >=, <>, #	(INTEGER, INTEGER) → BOOLEAN
Wertzuweisung	:=	INTEGER → INTEGER.

Aufzählungs- und Bereichstypen

MODULA-2 erlaubt zu den elementaren vordefinierten Typen weitere einzuführen. *Aufzählungstypen* (enumeration types) lassen sich wie folgt definieren:

```

TYPE Ampelfarbe = (rot, gelb, gruen);
   BOOLEAN      = (FALSE, TRUE);
                (* so in Modula-2 vordefiniert *)
   Tag           = (MO, TU, WE, TH, FR, SA, SU);

```

rot, TRUE, FR sind Literal-Konstanten ihres jeweiligen Typs. Die Typdefinition *Großmacht* = (USA, SU, CHINA) ist syntaktisch korrekt, aber mit den oben genannten nicht konsistent, denn SU wäre als Konstante nicht mehr einem bestimmten Typ zuzuordnen. Für Werte des gleichen Aufzählungstyps sind alle Vergleichsoperationen definiert (<, >, ≤, ≥, =, #).

Seien a und b Konstanten eines *skalaren*, d.h. elementaren Typs S mit $a \leq b$ (z.B. INTEGER, CHAR, ..., aber nicht REAL). Dann heißt $T = [a..b]$ ein *Bereichstyp* über dem *Grundtyp* S (Syntax nach Diagramm 16).

Beispiel:

```

Wochenende = [SA..SO];
StopFarben = [rot..gelb];
NatZahl    = INTEGER[0..MAXIMUM DIV 3];

```

Der eckigen Klammer kann der Basistyp vorangestellt werden (nur für INTEGER interessant, wenn diese trotz nichtnegativer Grenzen *nicht* als CARDINAL interpretiert werden sollen).

Typ-Anpassungen

Objekte verschiedenen Typs sind im allgemeinen *inkompatibel*; beispielsweise kann bei der INTEGER-Addition kein Operand vom Typ REAL oder CARDINAL sein, ebensowenig das Ergebnis. Darum ist die folgende Funktion falsch:

(* Programm P2.12 *)

```

PROCEDURE Summe (A, B: INTEGER) : REAL;
BEGIN (* Summe *)
  RETURN A + B; (* Fehler ! *)
END Summe;

```

Für bestimmte Sprachkonstrukte (Zuweisung, Parameterübergabe, Anwendung von Operatoren) sind aber gelegentlich *Typ-Anpassungen* notwendig. Nehmen wir an, daß die Funktion Ziegelmasse (Programm P2.8) neu die Masse von n Ziegeln berechnen soll, wobei n als formaler Parameter Anzahl vom Typ CARDINAL ist.

```

PROCEDURE Ziegelmasse
  (Laenge, Breite, Hoehe: REAL; Anzahl: CARDINAL) : REAL;

```

Anzahl darf nicht direkt als Multiplikand auftreten, sondern muß mittels einer *Typ-transfer-Funktion* gewandelt werden. Die RETURN-Anweisung lautet damit:

```

RETURN Laenge * Breite * Hoehe * Dichte * FLOAT (Anzahl);

```

Einige Typtransfer-Funktionen sind durch die nachfolgend angegebenen Prozedurköpfe beschrieben.

```

PROCEDURE FLOAT      (C: CARDINAL) : REAL;
PROCEDURE TRUNC      (R: REAL) : CARDINAL;
PROCEDURE INTEGER    (C: CARDINAL) : INTEGER;
PROCEDURE CARDINAL    (I: INTEGER) : CARDINAL;

```

Sie sind vordefiniert und damit ohne Import benutzbar wie die Absolutfunktion ABS, die INTEGER- oder REAL-Parameter erhält und ein betragsgleiches nichtnegatives Ergebnis vom gleichen Typ liefert. Programm P2.12 wäre also korrekt mit der RETURN-Anweisung RETURN FLOAT (A + B) .

Man beachte, daß es sich bei den oben angegebenen Typtransfer-Funktionen um *partielle* Funktionen handelt; ihr Ergebnis ist nur definiert, wenn es im Ergebnistyp darstellbar ist. Beispielsweise kann TRUNC(-2.5) ein beliebiges Resultat liefern oder den Abbruch der Programmausführung bewirken. Das Ergebnis von FLOAT ist nur exakt, wenn die Mantisse der REAL-Zahl ausreichend lang zur Darstellung des CARDINAL-Parameters ist. Es bleibt festzustellen, daß der Typtransfer in MODULA-2 allgemein stiefmütterlich behandelt ist. Auch die angegebenen Typtransfer-Funktionen gehören nicht zur Sprache und sind damit *implementierungsabhängig*.

2.2.5 Eingabevariablen

Solange wir alle Daten in Form von Konstanten codieren, ist jedes Programm auf einen bestimmten Datensatz fixiert. Wir wollen nun die Daten bis zur *Laufzeit* offenlassen. Daher müssen wir sie als *Variablen* deklarieren und *während* des Programmlaufs einlesen.

Streng genommen müssen wir unterscheiden zwischen dem *Variablen-Bezeichner*, also dem von uns frei gewählten Namen der Variablen, und der zugeordneten (uns nicht direkt sichtbaren) *Speicherzelle*, der Variablen im engeren Sinne. Meist ist die genaue Bedeutung aber aus dem Kontext sichtbar.

Für jede Variable ist eine *Deklaration* nötig. Durch diese werden dem MODULA-2-System Bezeichner und Typ mitgeteilt. Die Syntax ist den Diagrammen 13 (Declaration, der Zweig mit dem Wortsymbol VAR) und 37 (VariableDeclaration) zu entnehmen; als Typ kommen vorerst nur die vordefinierten Typen in Frage. Der Zwang zur Variablendeklaration und die daraus abgeleitete feste *Typbindung* (*strong typing*) ist ein wichtiges Kennzeichen moderner, vor allem imperativer Programmiersprachen.

Beispiel: VAR Heute : INTEGER;

Heute ist nur für INTEGER-Werte zugelassen, es kann z.B. nicht den Wert 1.5 annehmen (wohl aber -888). Für die *Eingabe* werden die Eingabeprozeduren ReadString, ReadInt oder ReadCard des Moduls InOut, oder die Prozedur ReadReal des Moduls RealInOut verwendet.

Beispiel: Für das Programm P2.5 Wochentag sähe das so aus (nur die Änderungen sind gezeigt):

```

MODULE Wochentag;                                     (* Programm P2.13 *)
FROM InOut IMPORT (* PROC *) ..., WriteString, ReadInt;
... (* die Konstantendefinition für "Heute" entfällt *)
VAR Heute : INTEGER;
BEGIN (* Wochentag *)
  WriteString ("Gib Nummer des heutigen Tages ein :");
  ReadInt (Heute);
  ...
END Wochentag.
```

Defensives Programmieren

Wie alle anderen technischen Artefakte ist ein Programm für einen bestimmten Anwendungsbereich vorgesehen, aber wie bei jenen wird dieser vielfach - meist unabsichtlich - überschritten. Das gilt besonders für Programme, die vom Benutzer direkt, d.h. interaktiv bedient werden können.

Ein Programm kann nun - völlig korrekt - ganz beliebig reagieren, wenn der Benutzer die *Vorschriften* nicht einhält (formal: wenn die *Vorbedingungen* für die Verwendung des Programms nicht erfüllt sind). Die Erfahrung lehrt aber, daß Benutzer eines Programms fast sicher "Fehler" im Sinne eines unerwarteten Verhaltens machen. Da ein undefiniertes Verhalten wenigstens bei häufigen Fehlern nicht akzeptabel ist, müssen Programme so verändert werden, daß die Anfangsbedingung entsprechend

gelockert werden kann, möglichst bis hin zum Idealfall, in dem sie in jedem Fall erfüllt ist. Ein Programm wird dadurch *robust(er)*.

Für die Benutzerschnittstelle eines Programms heißt das, daß jede Eingabe zunächst darauf untersucht werden muß, ob sie den *formalen* Anforderungen genügt, beispielsweise, ob eine "Zahl" auch tatsächlich aus Ziffern besteht.

Darum darf ein wirklich robustes Programm z.B. keine Zahl direkt mit ReadCard einlesen, denn die Eingabe eines Buchstabens oder einer negativen Zahl hat - maschinenabhängig - unabsehbare Folgen.

Bei der Codierung von robusten Eingabe-Anweisungen sind die folgenden *Regeln* zu beachten:

- vorher *Aufforderung* zur Eingabe ausgeben ("prompten"), möglichst mit Hinweis auf die korrekte Eingabeform,
- fehlerhafte Eingabe muß *sinnvoll* behandelt werden,
- *Ausweg* aus dem Programm anbieten für den Fall, daß der Benutzer beschließt, *keine* Eingabe zu machen.

Für die Kommunikation zwischen Teilen eines Programms (z.B. Prozeduren, Funktionen) gilt prinzipiell das gleiche: Eine Funktion ist umso robuster, je "mißtrauischer" sie ihre Parameter vor der Anwendung überprüft. In Analogie zum defensiven Fahren im Straßenverkehr spricht man dabei vom "*defensiven Programmieren*".

2.2.6 Rekursive Funktionen und Prozeduren

Allgemein bezeichnet man als *Rekursion* die Definition eines Problems, einer Funktion oder ganz allgemein eines Verfahrens "durch sich selbst". Wir nennen Algorithmen oder Programme im folgenden *rekursiv*, wenn sie Funktionen oder Prozeduren enthalten, die sich selbst direkt oder indirekt aufrufen. Ein Beispiel für Rekursion ist das Bild, das als Teil im Bild selbst vorkommt (realisierbar durch eine auf einen eigenen Monitor gerichtete Fernsehkamera).

Ein Beispiel einer rekursiven Prozedur ist die folgende Anweisung zur Zerlegung und zum Zusammenbau russischer Holzpuppen (Matrioschka-Puppen).

```

PROCEDURE betrachtePuppe (Puppe);                (* Programm P2.14 *)
(* Pseudocode für Prozedur zum Betrachten einer Matrioschka *)
BEGIN (* betrachtePuppe *)
  IF NOT massiv (Puppe) THEN
    oeffnePuppe (Puppe); (* gibt InhaltDerPuppe frei *)
    betrachtePuppe (InhaltDerPuppe); (* Rekursion *)
    schliessePuppe (Puppe);
  END (* IF *);
END betrachtePuppe;

```

Man bezeichnet eine solche programmähnliche, noch unvollständige, da nicht vollständig in einer Programmiersprache ausgeführte Formulierung als *Pseudocode*. Der

Ablauf dieser Prozedur ist für den Fall einer Puppe B1, die B2 und diese die massive B3 enthält, wie folgt:

```

Anfang betrachtePuppe (B1);
  oeffnePuppe (B1);           (* gibt B2 frei *)
  Anfang betrachtePuppe (B2);
    oeffnePuppe (B2);         (* gibt B3 frei *)
    Anfang betrachtePuppe (B3); (* ist massiv *)
    Ende betrachtePuppe (B3);
    schliessePuppe (B2);
  Ende betrachtePuppe (B2);
  schliessePuppe (B1);
Ende betrachtePuppe (B1);

```

Zum Zeitpunkt des Aufrufs `betrachtePuppe(B3)` existieren gleichzeitig drei *Inkarnationen* (verschiedene, noch nicht abgeschlossene Aufrufe) dieser Prozedur.

Die Rekursion kann man auch als ein Grundprinzip des Lebens bezeichnen, denn sie liegt der biologischen Vermehrung zugrunde. Dabei findet eine *unkontrollierte* Rekursion statt. In der Programmierung können wir nur mit einer *begrenzten* Rekursion arbeiten. Daher gehört jeder rekursive Funktionsaufruf in eine bedingte Anweisung, so daß er in speziellen Fällen *nicht* ausgeführt wird (*Abbruch der Rekursion*). Andernfalls endet die Ausführung des Programms (spätestens) wegen eines *Laufzeitfehlers*, denn jede Inkarnation benötigt zusätzlichen Speicherplatz.

Beispiele

Anhand dreier Beispiele (Fakultät, ggT und Türme von Hanoi) wollen wir die Verwendung rekursiver Funktionen und Prozeduren einüben.

1. Beispiel: Fakultät-Funktion

Die Fakultät-Funktion wird üblicherweise wie folgt rekursiv definiert:

$$\begin{aligned}
 0! &= 1 \\
 n! &= n \cdot (n - 1)! \quad \text{für alle } n \in \mathbb{N}
 \end{aligned}$$

Sie läßt sich auch genauso implementieren (außer, daß der Zahlenbereich eingeschränkt ist):

```

MODULE Fakultaet;                                     (* Programm P2.15 *)
(* Fakultaet einer vom Benutzer eingegebenen Zahl *)
(* Demonstration rekursiver Programme; *)
(* falsche Zeichen und zu große Zahlen werden *)
(* nicht abgefangen *)

FROM InOut IMPORT (* PROC *) WriteString, ReadInt,
                  WriteInt, WriteLn;

VAR EingabeZahl: INTEGER;

```



```

PROCEDURE FakRek (Argument: INTEGER) : INTEGER;
(* $RESULTAT: FakRek (Argument) = Argument ! *)
(* $VORBEDINGUNG: 0 <= Argument <= MaxArgument *)
(* Achtung, MaxArgument ist noch nicht berechnet *)
BEGIN (* FakRek *)
  IF Argument <= 1 THEN
    RETURN 1;
  ELSE
    RETURN Argument * FakRek (Argument-1);
  END (* IF *);
END FakRek;

BEGIN (* Fakultaet *)
  WriteString ('Gib natuerliche Zahl: ');
  ReadInt (EingabeZahl);
  WriteLn;
  IF EingabeZahl < 0 THEN
    WriteString ('Zahl '); WriteInt (EingabeZahl, 6);
    WriteString (' ist unzuessaessig. '); WriteLn;
  ELSE
    WriteInt (EingabeZahl, 6); WriteString ('! = ');
    WriteInt (FakRek (EingabeZahl), 6); WriteLn;
  END (* IF *);
END Fakultaet.

```

2. Beispiel: Größter gemeinsamer Teiler

Ein anderer bekannter rekursiver Algorithmus ist der von Euklid zur Berechnung des größten gemeinsamen Teilers (GGT). Ihm liegt die folgende Definition zugrunde:

Seien p und q gegeben, $p, q \in \mathbb{N}$ Dann gilt:

$$\text{GGT}(p, q) = \begin{cases} \text{GGT}(p-q, q), & \text{falls } p > q \\ p, & \text{falls } p = q \\ \text{GGT}(p, q-p), & \text{falls } p < q \end{cases}$$

Die folgende Funktion implementiert diesen Algorithmus. Hier wird die Form der IF-Anweisung mit ELSIF-Teil verwendet (vgl. Syntaxdiagramm 47).

```

PROCEDURE GGT (p, q: CARDINAL) : CARDINAL; (* Programm P2.16 *)
(* $FUNKTION: Berechnung des groessten gemeinsamen Teilers *)
BEGIN (* GGT *)
  IF p > q THEN RETURN GGT (p-q, q);
  ELSIF p < q THEN RETURN GGT (p, q-p);
  ELSE RETURN p;
  END (* IF *);
END GGT;

```

3. Beispiel: Die Türme von Hanoi

Als letztes ausführliches Beispiel für die Rekursion betrachten wir das bekannte Spiel "Türme von Hanoi". Es hat für vier Scheiben die in Abb. 2.14 gezeigte Grund-

stellung: auf dem linken Pfosten (ALPHA) liegen vier Scheiben, der Größe nach sortiert, die auf den mittleren Pfosten (OMEGA) zu transferieren sind.

Abb. 2.14 Grundstellung für vier Scheiben bei "Türme von Hanoi"

Dabei darf stets nur *eine* Scheibe bewegt werden und niemals darf eine Scheibe auf einer kleineren liegen. Natürlich dürfen die Scheiben nur auf die Pfosten gelegt werden.

Die *Lösungsstrategie* kann man wie folgt beschreiben: Das Problem wird allgemein für einen Turm der Höhe n gelöst, der von ALPHA nach OMEGA (via, d.h. mit Hilfe von DELTA) zu befördern ist. Dabei treten zwei Fälle auf:

- $n = 0$: gar nichts machen
- $n > 0$:
 - 1) Turm der Höhe $n-1$ von ALPHA nach DELTA bewegen (via OMEGA)
 - 2) Scheibe von ALPHA nach OMEGA legen
 - 3) Turm der Höhe $n-1$ von DELTA nach OMEGA bewegen (via ALPHA)

Damit ist das Problem durch Rekursion auf ein sehr einfaches Problem *reduziert*. Das entsprechende Programm sieht so aus:

```

MODULE Hanoi;                                     (* Programm P2.17 *)
(* Ausgabe der Zugfolge fuer die Tuerme von Hanoi *)
(* Demonstration rekursiver Prozeduren *)
FROM InOut IMPORT (* PROC *) WriteString, WriteLn,
                  WriteCard;
CONST Gesamthoehe = 4;
TYPE Hoehenbereich = [0..Gesamthoehe];
    Platz = (ALPHA, OMEGA, DELTA);

PROCEDURE bewegeTurm (Hoehe: Hoehenbereich;
                    vonPlatz, nachPlatz, ueberPlatz: Platz);
PROCEDURE DruckeZug (Start, Ziel: Platz);

PROCEDURE DruckePlatz (spezPlatz: Platz);
BEGIN (* DruckePlatz *)
  CASE spezPlatz OF
    | ALPHA: WriteString ('ALPHA');
    | OMEGA: WriteString ('OMEGA');
  
```



```

    | DELTA: WriteString ('DELTA');
  END; (* CASE *)
END DruckePlatz;

BEGIN (* DruckeZug *)
  WriteString ('Scheibe '); WriteCard (Hoehe,1);
  WriteString (' von '); DruckePlatz (Start);
  WriteString (' nach '); DruckePlatz (Ziel); WriteLn;
END DruckeZug;

BEGIN (* bewegeTurm *)
  IF Hoehe > 0 THEN
    bewegeTurm (Hoehe - 1,
                vonPlatz, ueberPlatz, nachPlatz);
    DruckeZug (vonPlatz, nachPlatz);
    bewegeTurm (Hoehe - 1,
                ueberPlatz, nachPlatz, vonPlatz);
  END (* IF *);
END bewegeTurm;

BEGIN (* Hanoi *)
  bewegeTurm (Gesamthoehe, ALPHA, OMEGA, DELTA);
END Hanoi.

```

Protokoll der Ausführung:

```

Scheibe 1 von ALPHA nach DELTA
Scheibe 2 von ALPHA nach OMEGA
Scheibe 1 von DELTA nach OMEGA
Scheibe 3 von ALPHA nach DELTA
Scheibe 1 von OMEGA nach ALPHA
Scheibe 2 von OMEGA nach DELTA
Scheibe 1 von ALPHA nach DELTA
Scheibe 4 von ALPHA nach OMEGA
Scheibe 1 von DELTA nach OMEGA
Scheibe 2 von DELTA nach ALPHA
Scheibe 1 von OMEGA nach ALPHA
Scheibe 3 von DELTA nach OMEGA
Scheibe 1 von ALPHA nach DELTA
Scheibe 2 von ALPHA nach OMEGA
Scheibe 1 von DELTA nach OMEGA

```

Es existieren folgende *Inkarnationen* von *bewegeTurm*, wenn die erste Zeile des obigen Protokolls gedruckt wird:

```

bewegeTurm (4, ALPHA, OMEGA, DELTA)
bewegeTurm (3, ALPHA, DELTA, OMEGA)
bewegeTurm (2, ALPHA, OMEGA, DELTA)
bewegeTurm (1, ALPHA, DELTA, OMEGA)

```

Die "offenen" Prozeduren, deren Ausführung unterbrochen ist, werden in einem sogenannten *Kellerspeicher* "eingefroren" und wieder "aufgetaut", wenn der darin enthaltene Prozeduraufruf abgeschlossen ist.

2.2.7 Nachteile funktional-rekursiver Programme

Leonardo Fibonacci hat in seinem 1202 erschienenen "Liber abaci" die Frage gestellt, wieviele Kaninchen-Pärchen es nach n Jahren gebe, wenn man im Jahr 1 mit einem Pärchen beginnt, jedes Pärchen vom zweiten Jahr an ein weiteres Pärchen Nachwuchs hat und die Kaninchen eine unendliche Lebensdauer haben. Offenbar entsteht die folgende Zahlenreihe:

Jahr	1	2	3	4	5	6	7	8	...
Zahl der Pärchen	1	1	2	3	5	8	13	21	...

Diese Art von Wachstum gibt es auch in vielen anderen Bereichen, so daß die sogenannten *Fibonacci-Zahlen* allgemeine Bedeutung haben. Sie lassen sich wie folgt definieren:

$$\begin{aligned} \text{Fibonacci}(n) &= 1 && \text{für } n = 1, 2 && \text{und} \\ \text{Fibonacci}(n) &= \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) && \text{für } n > 2. \end{aligned}$$

Die Definition ist rekursiv, und entsprechend sieht die einfachste Lösung einer Funktionsprozedur aus :

```

(* Programm P2.18 *)
PROCEDURE Fibonacci (Argument: INTEGER) : INTEGER;
BEGIN (* Fibonacci *)
  IF Argument <= 2 THEN
    RETURN 1;
  ELSE
    RETURN Fibonacci (Argument-1) + Fibonacci (Argument-2);
  END (* IF *);
END Fibonacci;
```

Ein Algorithmus heißt dann und nur dann *effektiv*, wenn er unter allen vorgesehenen Bedingungen nach endlich vielen Schritten das korrekte Ergebnis liefert. In diesem Sinne ist die Funktion Fibonacci effektiv (die Begrenzung des Zahlenbereichs wird hier ignoriert).

Neben der Effektivität spielt aber auch die *Effizienz* (Aufwand) eines Algorithmus eine entscheidende Rolle. Wir behandeln dieses Thema zwar noch ausführlich in Abschnitt 6.1, wollen das Problem in Verbindung mit einer funktional-rekursiven Prozedur wie Fibonacci aber schon hier aufgreifen.

Bei Aufrufen der Funktion Fibonacci werden (außer, wenn der Parameter 1 oder 2 ist) stets die gleichen Schritte ausgeführt. Der Aufwand - und damit die notwendige Rechenzeit - ist also etwa *proportional* der Zahl der Aufrufe, die hier mit c bezeichnet werden soll. Wir wollen nun abschätzen, wie groß c ist in Abhängigkeit vom Argument n (hier als Index von c), mit dem Fibonacci aus dem Hauptprogramm aufgerufen wird.

Offenbar gilt: $c_1 = c_2 = 1$
 Für $n > 2$: $c_n = 1 + c_{n-1} + c_{n-2}$,
 Bei $n > 3$: $c_{n-1} = 1 + c_{n-2} + c_{n-3}$,
 also $c_{n-2} = c_{n-1} - 1 - c_{n-3}$.

Wir setzen daraus c_{n-1} oder c_{n-2} in die Gleichung für c_n ein und erhalten:

$c_n = 2 + 2 c_{n-2} + c_{n-3} > 2 c_{n-2} >$
 $> 2^2 c_{n-4} > 2^3 c_{n-6} > \dots > 2^n \text{ DIV } 2^{-1} c_2$
 also $c_n > 2^n \text{ DIV } 2^{-1}$ (d.h. $2^n \text{ DIV } 2^{-1}$ ist eine Untergrenze für c_n)
 oder $c_n = 1 + c_{n-1} + c_{n-2} = 2 c_{n-1} - c_{n-3} < 2 c_{n-1}$
 also $c_n < 2^n$ (d.h. 2^n ist eine Obergrenze für c_n).

Die gezeigte Funktion Fibonacci erfordert offenbar einen mit n *exponentiell* wachsenden Aufwand, was praktisch schon für Parameter etwa in der Größenordnung von 40 eine Laufzeit von vielen Stunden bedeutet. Folglich ist sie *nicht effizient* für große n . Eine Funktion (oder ein Programm) nennen wir nur dann effizient, wenn das Ergebnis mit einem Aufwand erreicht wird, der innerhalb der vorgegebenen Grenzen liegt (bei Interaktion typisch einige Sekunden).

Fazit: In der Informatik wird zwischen Effektivität und Effizienz strenger unterschieden als in der Umgangssprache!

2.3 Iterative Programme

Wir haben uns bisher nur mit funktionalen Programmen beschäftigt, Programmen also, die Variablen nicht oder nur zur Eingabe verwenden. Ein Algorithmus, der Zwischenergebnisse in *Variablen* ablegt, heißt *nicht-funktional*. Er heißt *iterativ*, wenn bestimmte Abschnitte des Algorithmus innerhalb einer einzigen Ausführung mehrfach durchlaufen werden.

Iteration und Rekursion sind Prinzipien, die oft unpassend als Alternativen für die Programmkonstruktion bezeichnet werden. Iterative Programme sollten aber nur als alternativ zu funktionalen Programmen gesehen werden, denn sie können nicht funktional sein, da der Fortschritt der Iteration durch ein Zwischenergebnis gespeichert werden muß. Die Begriffe "funktional" und "rekursiv" werden daher oft miteinander verwechselt, obwohl eine klare Unterscheidung sinnvoll ist.

2.3.1 Wertzuweisungen und Referenz-Parameter

Wertzuweisung

Wir führen nun als wesentliches Merkmal der nicht-funktionalen Programmierung die *Wertzuweisung* ein. Ihre allgemeine Form ist " $V := E$ ". Darin ist V ein Variablenname und E ein Ausdruck, beispielsweise eine Zahl, eine Variable oder ein arithmetischer Ausdruck aus Zahlen, Variablen und Operatoren.

Es handelt sich bei " $:=$ " (lies: "wird" oder "erhält den Wert", *nicht* "gleich") um einen *unsymmetrischen* Operator; der Ausdruck auf der rechten Seite wird *ausgewertet* und das Resultat anschließend der Variablen links *zugewiesen*. Noch klarer im Sinne des Ablaufs, den wir traditionell mit einer Bewegung von links nach rechts assoziieren, wäre eine Syntax, bei der Zuweisungsoperator und Variable auf der *rechten* Seite stehen. Diese Sichtweise hat sich aber nicht durchsetzen können.

Die Verwendung des einfachen Gleichheitszeichens in älteren Programmiersprachen (FORTRAN, C usw.) ist irreführend, da dieses in der Mathematik eine ganz andere Bedeutung hat und in dieser Bedeutung auch gebraucht wird (in FORTRAN mußte darum für das eigentliche " $=$ " ein Operator ".EQ." eingeführt werden).

In MODULA-2 müssen Ausdruck und Variable *typkompatibel* sein, d.h. das Resultat muß im Wertebereich der Variablen liegen.

Beispiel:

Nach der Deklaration

```
VAR x, y, z: CARDINAL;
```

wären einige (korrekte und inkorrekte) Wertzuweisungen für x:

```
x := 0; x := MAX (CARDINAL); x := y; (* sicher richtig *)
x := -1; x := MAX (CARDINAL)+1; x := -y-1; (* sicher falsch *)
x := y+z; x := z-y; (* richtig oder falsch, *)
(* je nach Wert von y, z *)
```

In vielen Fällen kann schon der Übersetzer Inkompatibilitäten feststellen; sonst kommt es bei der Programmausführung zu einem *Laufzeitfehler*.

Referenz-Parameter

Die bisher eingeführten Prozedur-Parameter werden auch als *Wert-Parameter* bezeichnet, denn das Unterprogramm erhält nur den Wert zum Zeitpunkt des Aufrufs mitgeteilt ("*call-by-value*"). In den folgenden Beispielen kommen auch andere Parameter vor, die in der formalen Parameterliste durch VAR (für "variable") gekennzeichnet sind, z.B. in:

```
PROCEDURE FibProc (Arg: CARDINAL; VAR Fib, FibMin1 : CARDINAL);
```

Hier handelt es sich um sogenannte *Referenz-Parameter*; bei ihrer Übergabe an ein Unterprogramm erhält dieses die Variable zur freien Verwendung. Technisch wird das durch Übergabe ihrer Speicheradresse realisiert ("*call-by-reference*"). Damit ist eine *Kommunikation* in beiden Richtungen möglich. Wird im Beispiel der Referenz-Parameter Fib durch FibProc verändert, so hat die Variable, die als konkreter Parameter dafür gesetzt worden war, nach dem Aufruf einen anderen Wert als vorher. Beim Wert-Parameter Arg ist dies nicht unmöglich.

In der Programmiersprache Ada werden alle drei möglichen Kommunikationsarten unterschieden, nämlich

- *in* für die Wert-Parameter,
- *out* für Ergebnis-Parameter (gibt es in MODULA-2 nicht) und
- *in out* für die Parameter, die sowohl Wert- als auch Ergebnis-Parameter sind.

Durch Wert-Parameter sind keine Änderungen von konkreten Parametern durch den aufgerufenen Programmteil möglich. Darum sind sie grundsätzlich vorzuziehen. Referenz-Parameter sollten nur verwendet werden, wenn

- darin Prozedur-Ergebnisse *übergeben* werden oder
- Kopieren *nicht* möglich ist (bei gewissen Datenstrukturen) oder
- Kopieren zu *ineffizient* ist (bei großen Datenstrukturen).

Funktionsprozeduren sollten keine Referenz-Parameter zur Rückgabe von Ergebnissen enthalten, denn deren Effekt ist meist schwer zu erkennen und damit oft eine Fehlerquelle.

Es gilt also als *Regel*: Eine Prozedur liefert ihr Ergebnis entweder durch ihren Namen *oder* durch Referenz-Parameter, aber nicht gemischt. Das erste Prinzip (*Funktionsprozedur*) ist vorzuziehen, aber (in MODULA-2) nur anwendbar, wenn sich das Ergebnis auf einen einzelnen Wert eines elementaren Typs beschränkt.

2.3.2 Gültigkeitsbereich und Lebensdauer

Der *Gültigkeitsbereich* ("scope") eines Bezeichners ist derjenige *statische* Teil des Programms, in dem dieser Bezeichner in exakt gleicher Bedeutung verwendet werden darf. Es handelt sich also um einen statischen Begriff, zu dessen Erörterung keine Überlegungen zum Ablauf des Programms erforderlich sind. Der Begriff *Lebensdauer* bezeichnet die *dynamische* Existenz eines Objekts während des Programmablaufs.

Es ist wichtig, beide Begriffe klar zu unterscheiden: Der Gültigkeitsbereich bezieht sich stets auf den Bezeichner; er wird durch den Übersetzer kontrolliert. Die Lebensdauer bezieht sich dagegen auf den zur *Laufzeit* belegten Speicherplatz; dies betrifft den Maschinencode des Programms und das *Laufzeitsystem*. Da Konstanten und Typen keinen Speicherplatz belegen, haben sie auch keine Lebensdauer.

Gültigkeitsbereich

Im folgenden sei P ein Modul oder eine Prozedur. Seien darin eine Konstante X und eine Variable Y definiert, z.B.

```
PROCEDURE P ...; CONST X = 85; VAR Y: REAL; ...
```

X und Y können in der festgelegten Bedeutung in einem gewissen Bereich von P, dem Gültigkeitsbereich, verwendet werden. Der Gültigkeitsbereich erstreckt sich über den ganzen Block von P (also die gesamte Prozedur oder das gesamte Modul ohne Prozedur- bzw. Modulkopf). Davon ausgenommen sind alle Deklarationen, die vor der Einführung von X bzw. Y stehen. Die gleichen Regeln gelten auch für Typen und Prozeduren. Damit kann auch *wechselseitige Rekursion* (gegenseitig rekursiver Aufruf zweier Prozeduren, vgl. P2.19a) implementiert werden.

Bei der Überlappung von Gültigkeitsbereichen kann es überraschende Effekte geben. Beispielsweise wird das Programm P2.19c auf *einem* MODULA-2-Compiler ohne Fehlermeldung übersetzt und liefert als Ergebnis die Zahlen 5, 6, 7, 10 und 5. Ein *anderer* Compiler "stört sich" an der Vereinbarung von a mittels a, hat aber im übrigen das gleiche Ergebnis.

```

MODULE M;                                     (* Programm P2.19c *)
(* Grenzfaelle des GBs *)
FROM InOut IMPORT (* PROC *) WriteInt, WriteLn;
CONST a = 5;

PROCEDURE P;
CONST u = a+1; a = a+2; v = a+3;
BEGIN (* P *)
  WriteInt (u, 3); WriteInt (a, 3); WriteInt (v, 3);
END P;

BEGIN (* M *)
  WriteInt (a, 3); P; WriteInt (a, 3); WriteLn;
END M.

```

Man beachte, daß MODULA-2-Übersetzer grundsätzlich nicht "mitdenken", also nicht aufgrund der Typdefinition entscheiden, welche Bedeutung gemeint ist. Die Zuordnung erfolgt *schematisch* aufgrund der Gültigkeitsbereiche; ist der Typ dann nicht konsistent, so meldet der Compiler einen Fehler.

Lebensdauer

Durch Ausführung einer Prozedur P entsteht eine *Inkarnation* ("Aktivierung"). Ist P ein Modul, so gelten etwas andere, weiter unten vorgestellte Regeln. Zur Inkarnation gehören *dynamisch*:

- ein *Ausführungspunkt* (also ein Zeiger auf den gerade auszuführenden oder ausgeführten Befehl)
- *Speicherplätze* für alle Bezeichner von Variablen und Wert-Parameter
- *Bezüge* auf die konkreten Referenz-Parameter.

Diese Informationen existieren bis zum Ende der Ausführung von P.

Beispiel:

```
PROCEDURE Example (ch: CHAR; VAR x:INTEGER); VAR y: REAL; ...
```

Durch den Aufruf Example ('a', z) entsteht eine Inkarnation von Example mit Speicherplatz für ch und y und - unter dem lokalen Bezeichner x - einem Bezug (einer Referenz) auf z. Nach Abschluß von Example werden diese Speicherplätze wieder freigegeben. Handelt es sich bei P um ein Modul, so wird der *Speicherplatz* für die

Variablen permanent für die gesamte Laufzeit des Programms reserviert. Eine eigentliche Inkarnation wird nur von dem Rumpf des Moduls gebildet; dieser hat weder Variablen noch Parameter.

Bemerkungen:

- Zu irgendeinem Zeitpunkt existieren i.a. neben der Inkarnation des ablaufenden Moduls Inkarnationen verschiedener Prozeduren (inkl. Funktionen), bei rekursiven Aufrufen auch mehrere der gleichen Prozedur.
- Nur in der jüngsten aller existierenden Inkarnationen wandert der Ausführungspunkt weiter; diese wird als erste beendet.
- Die Lebensdauer einer Variablen ist identisch mit der Existenz der zugehörigen Prozedur-Inkarnation.
- Zu Beginn der Lebensdauer ist der Wert einer Variablen undefiniert, darf also nicht verwendet werden.

Beispiel (zur Analyse von Gültigkeitsbereich und Lebensdauer, vgl. Abb. 2.15):

```

MODULE LDUGB;                                     (* Programm P2.20a *)
FROM InOut IMPORT (* PROC *) WriteInt, WriteLn;
VAR i, j: INTEGER;

  PROCEDURE Summe (k: INTEGER) : INTEGER;
  VAR j: INTEGER;
  BEGIN (* Summe *)
    IF i < 10 THEN j := 10 ELSE j := 1 END;
    RETURN j + k;
  END Summe;

  PROCEDURE Drucken (j: INTEGER);
  VAR i: INTEGER;
  BEGIN (* Drucken *)
    IF i <> j THEN i := Summe(j) END;
    WriteInt (Summe(i), 3); WriteLn;
  END Drucken;

BEGIN (* LDUGB *)
  i := 17; j := 4;
  Drucken (Summe(j));
  Drucken (Summe(i));
END LDUGB.

```

Zunächst sollen die *Gültigkeitsbereich* analysiert werden. Dazu markieren wir die definierenden Auftreten der Bezeichner (durch Fettdruck) und kennzeichnen, wo derselbe Bezeichner in verschiedenen Bedeutungen verwendet wird, den Zusammenhang zwischen Definition und Anwendung durch Indizes. Natürlich sind diese Indizes nicht Teil des Programms!


```

MODULE LDUGB;                                     (* Programm P2.20b *)
FROM InOut IMPORT (* PROC *) WriteInt, WriteLn;
VAR i1, j1: INTEGER;

PROCEDURE Summe (k: INTEGER) : INTEGER;
VAR j2: INTEGER;
BEGIN (* Summe *)
  IF i1<10 THEN j2:= 10 ELSE j2:=1 END;
  RETURN j2 + k;
END Summe;

PROCEDURE Drucken (j3: INTEGER);
VAR i2: INTEGER;
BEGIN (* Drucken *)
  IF i2 <> j3 THEN i2:= Summe(j3); END;
  WriteInt (Summe(i2), 3); WriteLn;
END Drucken;

BEGIN (* LDUGB *)
  i1:= 17; j1:= 4;
  Drucken (Summe(j1));
  Drucken (Summe(i1));
END LDUGB.

```

Nun folgt die Analyse der Lebensdauer. Dabei müssen wir den *Ablauf* des Programms zugrundelegen. In Abb. 2.15 läuft die Zeitachse von oben nach unten. Die Ausführung des Modul-Rumpfs (LDUGB) entspricht der einer Prozedur.

Die *Laufzeitfehler* (durch lesenden Zugriff auf eine nicht initialisierte Variable) können natürlich verschiedene Effekte haben; meist wird das Programm - leider - ohne Meldung weiterlaufen und den zufällig in der betreffenden Speicherzelle vorhandenen Wert verwenden. So ist es auch in Abb. 2.15. Übrigens wird dann dieser Wert gleich noch einmal gelesen.

Legende:

Inkarnation (Aktivierung) einer Prozedur

Lebensdauer einer Variablen oder eines Parameters mit Besetzung und Abfrage (schreibendem/lesendem Zugriff)

Informationsfluß (Parameter-Übergabe, Funktionsresultat).

Abb. 2.15 Graphische Darstellung der Lebensdauern beim Ablauf von LDUGB

Mehrfachverwendung von Bezeichnern

Das Ergebnis des Programms P2.21a soll ermittelt werden.

```

MODULE ParamTest;                                     (* Programm P2.21a *)
FROM InOut IMPORT (* PROC *) WriteInt, WriteLn;
VAR a, b, c: INTEGER;

  PROCEDURE ParProc (a: INTEGER; VAR b:INTEGER);
  BEGIN (* ParProc *)
    b:= a + c;
    c:= b + 4;
    WriteInt (a+b+c, 3); WriteLn;
  END ParProc;

BEGIN (* ParamTest *)
  a:= 3; b:= 2; c:= 1;
  ParProc (c, c);
END ParamTest.

```

Folgende Punkte sind zu berücksichtigen:

- In ParProc bezeichnet a den Wert-Parameter, der beim Aufruf mit c, also 1 initialisiert ist.
- c ist global, also auch in der Prozedur gültig.
- Ein *Referenz-Parameter* (b innerhalb der Prozedur) liegt auf der gleichen Speicherstelle wie die aufrufende Variable. Darum gilt an jedem Punkt der Prozedur $b = c$.

Das Programm liefert als Resultat 13. Die konfuse Kommunikation dieses Beispiels ist natürlich ein sehr schlechter Stil!

Mehrfachverwendung von Bezeichnern und Rekursion

Wie bei Mehrfachverwendung von Bezeichnern, aber mit zusätzlicher Zeile vor der Ausgabe (Werte: 16, 26):

```

                                                                (* Programm P2.21b *)
PROCEDURE ParProc (a: INTEGER; VAR b:INTEGER);
BEGIN (* ParProc *)
  b:= a + c;
  c:= b + 4;
  IF a # 0 THEN ParProc (0, a) END;
  WriteInt (a+b+c, 3); WriteLn;
END ParProc;

```

Terminologie

Was durch einen vordefinierten oder frei wählbaren Bezeichner eingeführt ist (also Modul, Prozedur, Konstante, Typ, Variable, Parameter), wird als *Objekt* bezeichnet. Ausgenommen sind also Operatoren (z.B. +, DIV), Wortsymbole (z.B. VAR, END) und Begrenzer (:).

Ein Objekt, das im Block B deklariert ist, heißt *lokal in B*. Ein Bezeichner, der auf Modulebene deklariert ist, heißt *global* (ebenso das zugeordnete Objekt). Ist ein Bezeichner in B gültig, aber nicht lokal, so heißt er *global relativ zu B*.

Variablen, die lokal in B sind, werden bezüglich ihres Gültigkeitsbereich durch B, bezüglich ihrer Lebensdauer durch die Aktivierung von B begrenzt. Das gleiche gilt für *Wert-Parameter*, nur werden diese durch den Aufruf initialisiert. Bei *Referenz-Parametern* gilt dies nur für den Bezeichner; eine Lebensdauer hat der Referenz-Parameter nicht, sondern nur die Variable, durch die er konkret besetzt wird.

Funktionswerte (Prozedur-Ergebnisse) können nur lesend verwendet, also nicht durch eine Wertzuweisung oder eine Verwendung als Referenz-Parameter verändert werden.

Hinweise zum Programmierstil

Programme sollten mit kleinstmöglichem Aufwand korrigier- und modifizierbar sein. Ein Ansatz zur Erlangung dieses Ziels ist eine *hohe Lokalität* durch enge Gültigkeitsbereiche. Größtmögliche Lokalität ist daher vorrangiges Ziel einer guten Programmierung! Ein Programm sollte dafür folgende Merkmale aufweisen:

- Die auftretenden Programmeinheiten (Prozeduren, Funktionen, Hauptprogramm) sind *überschaubar*.
- Die Objekte sind so *lokal* wie möglich definiert, jeder Bezeichner hat nur eine einzige, bestimmte Bedeutung.
- Die *Kommunikation* zwischen Programmeinheiten erfolgt vorzugsweise über eine möglichst kleine Anzahl von Parametern, nicht über globale Variablen.

Wie gezeigt wurde, besteht zwischen Gültigkeitsbereich und Lebensdauer eine enge Beziehung: Alle sichtbaren Bezeichner gehören zu existierenden Objekten und alle existierenden Objekte sind durch Bezeichner adressierbar, sofern diese nicht verdeckt sind. Dieser Zusammenhang besteht nicht von Natur aus, sondern ist eine charakteristische Eigenschaft *block-orientierter* Sprachen.

2.3.3 Anweisungen zur Iteration

Prinzipiell gibt es drei verschiedene Möglichkeiten zur Konstruktion einer Iteration:

- Schleife mit *vorheriger* Prüfung (WHILE)
- Schleife mit *nachfolgender* Prüfung (REPEAT)
- Schleife *ohne* Prüfung (LOOP),

die nachfolgend vorgestellt werden.

Schleife mit vorheriger Prüfung

Die allgemeine Form der Schleife mit vorheriger Prüfung, die WHILE-Schleife oder auch "ablehnende Schleife", ist (mit Bedingung C und Anweisung S):

```
WHILE C DO S END
```

Die Bedeutung dieser Anweisung läßt sich rekursiv mit Hilfe der bedingten Anweisung definieren, sie ist gleichbedeutend mit:

```
IF C THEN S; WHILE C DO S END; END
```

Die Lauschleife

```
FOR cv := exp1 TO exp2 DO S END
```

ist eine spezielle Form der WHILE-Schleife. Wir verwenden sie immer dann, wenn die Zahl der Iterationen schon vor Eintritt in die Schleife bekannt ist. Sie hat (mit dem neuen Bezeichner h2) die gleiche Bedeutung wie

```
cv := exp1; h2 := exp2;
WHILE cv <= h2 DO S; INC (cv); END;
```

In der Schleife kann auch ein *Inkrement* oder *Dekrement* angegeben werden:

```
FOR cv := exp1 TO exp2 BY exp3 DO S END
```

Dies hat (mit den neuen Bezeichnern h2 und h3) die Bedeutung

```
cv := exp1; h2 := exp2; h3 := exp3;
IF h3 > 0 THEN
  WHILE cv <= h2 DO S; cv := cv + h3; END
ELSE
  WHILE cv >= h2 DO S; cv := cv + h3; END
END (* IF *)
```

Für die FOR-Schleife gelten dabei aber einige *Einschränkungen*:

- cv darf weder Komponente einer Datenstruktur noch Parameter sein.
- Innerhalb der Schleife (Anweisung S) muß cv wie eine *Konstante* behandelt werden, darf also nicht auf der linken Seite einer Wertzuweisung oder als Referenz-Parameter stehen und damit manipulierbar sein. Das gilt natürlich auch für Prozeduren, die in diesem Bereich aufgerufen werden. Man sollte jedoch nicht darauf vertrauen, daß der Compiler oder das Laufzeitsystem solche Fehler erkennt.

- Fehlt das In- oder Dekrement, so darf cv auch Variable eines Aufzählungstyps sein. Andererseits entsteht kein Laufzeitfehler, wenn cv bei der letzten In-/Dekrementierung seinen Wertebereich verläßt. Man beachte, daß nach Verlassen der Schleife cv keinen definierten Wert hat!

Schleife mit nachfolgender Prüfung

Bei der Schleife mit nachfolgender Prüfung oder REPEAT-Schleife findet eine erste Ausführung statt, ohne daß die Bedingung (in diesem Fall eine *Abbruch*-Bedingung) geprüft wird (daher auch "annehmende Schleife"). Syntax:

```
REPEAT S1; S2; S3; ... ; Sn UNTIL C
```

Schleife ohne Prüfung

Die Schleife ohne Prüfung, die LOOP-Schleife, wird ausgeführt, bis sie durch Erreichen einer darin enthaltenen EXIT-Anweisung verlassen wird. Syntax:

```
LOOP S1; S2; ... ; Sn END
```

Hinweise zur Wahl des Iterationskonstrukts

REPEAT...UNTIL ist mit Vorsicht zu verwenden, denn die Anweisung in der Schleife wird mindestens einmal durchlaufen. Sie ist nur sinnvoll, wenn die Bedingung erst durch S entsteht (z.B. Eingabe mit Wiederholung im Fehlerfall). REPEAT ... UNTIL kommt vor allem dann in Frage, wenn die zu prüfende Bedingung erst in der Schleife entsteht.

Beispiel (Eingabe mit Prüfung auf falschen Zahlenwert durch WHILE):

```
WriteString ('Gib zweistellige Zahl: '); (* Programm P2.22a *)
ReadCard (Zahl); WriteLn;
WHILE (Zahl<10) OR (Zahl>99) DO
  WriteString (' Gib zweistellige Zahl: '); ReadCard (Zahl);
END (* WHILE *);
```

läßt sich eleganter formulieren durch:

```
REPEAT (* Programm P2.22b *)
  WriteString ('Gib zweistellige Zahl: '); ReadCard (Zahl);
UNTIL (Zahl>=10) AND (Zahl<=99);
```

Die Schleife ohne Prüfung macht aber die REPEAT-UNTIL-Konstruktion meist überflüssig; im Beispiel oben gestattet sie auch (wie die WHILE-Lösung) die Ausgabe einer Fehlermeldung:


```

LOOP                                     (* Programm P2.22c *)
  WriteString ('Gib zweistellige Zahl: '); ReadCard (Zahl);
  IF (Zahl>=10) AND (Zahl<=99) THEN EXIT END;
  WriteString (' *** Fehler in der eingegebenen Zahl !');
  WriteLn;
END (* LOOP *);

```

Wenn es darum geht, einen vorher definierten, lückenlosen Bereich eines skalaren Typs zu durchlaufen, so ist die FOR-Schleife angemessen. In allen anderen Fällen ist die allgemeinere WHILE-Schleife (oder in speziellen Fällen die REPEAT-Schleife) vorzuziehen.

2.3.4 Iterative und rekursive Lösungen

Nachfolgend werden Vor- und Nachteile verschiedener iterativer und rekursiver Lösungen anhand von vier verschiedenen Programmen am Beispiel der Fibonacci-Zahlen aufgezeigt:

- eine funktional-rekursive Lösung (A)
- iterative Lösungen mit (B) bzw. ohne (C) Umspeichern
- eine rekursive nicht-funktionale Lösung (D).

A. Funktional-rekursive Lösung

Die bereits vorgestellte Funktion Fibonacci ist hier für das durchgängige Rahmen-Modul Fibonacci leicht verändert: statt INTEGER wird CARDINAL verwendet, und die oberste Grenze für das Argument sei (durch Berechnung mit einem anderen Programm) vorgegeben. Die Prozedur FibFct1 in P2.23 stellt eine rein funktional-rekursive Lösung dar.

```

MODULE Fibonacci;                                     (* Programm P2.23 *)
(* berechnet Fibonacci-Zahl einer *)
(* natuerlichen Zahl bis MaxArg *)
FROM InOut IMPORT (* PROC *) WriteString, WriteLn,
                  ReadCard, WriteCard;
CONST MaxArg = 24; (* obere Grenze fuer das Argument *)
VAR Eingabe : CARDINAL;

PROCEDURE FibFct1 (Arg: CARDINAL) : CARDINAL;
(* funkt.-rekursiv *)
BEGIN (* FibFct1 *)
  IF Arg <= 1 THEN RETURN Arg;
  ELSE RETURN FibFct1 (Arg - 1) +
              FibFct1 (Arg - 2);
  END (* IF *);
END FibFct1;

```

```

BEGIN (* Fibonacci *)
  REPEAT
    WriteString ('Argument: '); ReadCard (Eingabe); WriteLn;
    UNTIL (Eingabe <= MaxArg);
    WriteString ('Fibonacci-Zahl ('); WriteCard (Eingabe, 3);
    WriteString (') = '); WriteCard (FibFct1 (Eingabe), 6);
    WriteLn;
  END Fibonacci.

```

Die Funktion FibFct1 ist hier elegant - und damit knapp - formuliert. In diesem Sinne ist der Algorithmus optimal. Wie aber schon im Abschnitt 2.2.7 gezeigt wurde, ist diese Lösung für große Argumente sehr ineffizient. Die (bei der hier geltenden Obergrenze ausgeschlossene) Berechnung der Fibonacci-Zahl von 30 (= 832.040) erfordert 1.664.079 Funktionsaufrufe (nämlich, wie man durch Induktion leicht zeigen kann, $2 \cdot \text{Fib}(30) - 1$) und dauert auf einem Rechner mit durchschnittlicher Leistungsfähigkeit mehrere Sekunden, bei Argumenten ab etwa 40 schon Stunden bis Tage. Wir ersetzen daher die rekursive Funktion durch eine iterative Lösung.

B. Iterative Lösung mit Umspeichern

Die Lösung in P2.24 folgt dem Verfahren, das man anwendet, wenn man die Fibonacci-Zahlen "von Hand" berechnet. Die jeweils interessanten Werte (d.h. die beiden zuletzt berechneten Fibonacci-Zahlen) werden in den Variablen Fib und FibMin1 gespeichert. Bei jeder Iteration wird eine weitere Fibonacci-Zahl berechnet und die genannten Variablen mit Hilfe des Zwischenspeichers FibNew umgespeichert.

```

(* Programm P2.24 *)
PROCEDURE FibFct2 (Arg: CARDINAL) : CARDINAL;
(* Iterative Variante ohne Feld *)
VAR Index, Fib, FibMin1, FibNew: CARDINAL;
BEGIN (* FibFct2 *)
  IF Arg = 0 THEN
    RETURN 0;
  ELSE
    FibMin1 := 0; Fib := 1;
    FOR Index := 2 TO Arg DO
      FibNew := Fib + FibMin1; FibMin1 := Fib; Fib := FibNew;
    END (* FOR *);
    RETURN Fib;
  END (* IF *);
END FibFct2;

```

C. Iterative Lösung ohne Umspeichern

Wenn man ohne Umspeichern (und damit ohne FibNew) auskommen will, kann man die Lösung in P2.25 wählen, bei der die Speicherung der interessanten Werte in einem sogenannten *Ringpuffer* (hier durch die beiden Variablen Ring0 und Ring1

realisiert) erfolgt. Man beachte aber, daß eine solche Lösung gut kommentiert sein muß, um verständlich zu sein.

```

(* Programm P2.25 *)
PROCEDURE FibFct3 (Arg: CARDINAL) : CARDINAL;
(* Iterative Variante mit Ringpuffer *)
TYPE IndexRange = [0..1];
VAR Ring0, Ring1 : CARDINAL;
    Pos          : CARDINAL;
BEGIN (* FibFct3 *)
    Ring0 := 0; Ring1 := 1;          (* Initialisierung des Rings *)
    FOR Pos := 2 TO Arg DO          (* Berechnung *)
        IF Pos MOD 2 = 0 THEN
            Ring0 := Ring0 + Ring1;
        ELSE
            Ring1 := Ring0 + Ring1;
        END (* IF *);
    END (* FOR *);
    IF Arg MOD 2 = 0 THEN
        Return Ring0;
    ELSE
        RETURN Ring1;
    END (* IF *);
END FibFct3;

```

D. Rekursive nicht-funktionale Lösung

Der große Gewinn an Effizienz zwischen der funktional-rekursiven Lösung und den iterativen Lösungen wird in der Regel dem Verzicht auf die Rekursion zugeschrieben. Das ist aber, wie die folgende *effiziente* rekursive Lösung in P2.26 zeigt, falsch. Tatsächlich gewinnen wir die höhere Effizienz durch die Verwendung von Variablen (und Ergebnis-Parametern), also durch den Verzicht auf funktionale Programmierung.

Dazu wird anstelle der Funktion eine Prozedur verwendet, die *zwei* Werte liefert. FibMin1 wird *einmal* berechnet, aber *zweimal* verwendet, nämlich direkt und als Ergebnis-Parameter. Damit steigt der Aufwand nur noch linear mit dem Argument, und die Fibonacci-Zahl von 30 kann mit 29 Prozeduraufrufen im Millisekundenbereich ermittelt werden. Diese Lösung ist aber offensichtlich komplizierter als die iterativen Lösungen und man würde in der Praxis i.a. darauf verzichten.

```

(* Programm P2.26 *)
PROCEDURE FibFct4 (Arg: CARDINAL) : CARDINAL;
(* Nicht-funktionale rekursive Variante *)

PROCEDURE FibProc (Arg: CARDINAL; VAR Fib, FibMin1: CARDINAL);
(* Liefert fuer Arg >= 0 in Fib die Fibonacci-Zahl *)
(* von Arg, in FibMin1 die von Arg-1. Fuer Arg = 0 *)
(* ist FibMin1 undef. *)

```

```

VAR FibMin2 : CARDINAL;
BEGIN (* FibProc *)
  IF Arg <= 1 THEN
    Fib := Arg; FibMin1 := 0;
  ELSE
    FibProc (Arg - 1, FibMin1, FibMin2);
    Fib := FibMin1 + FibMin2;
  END (* IF *);
END FibProc;

VAR FibRes1, FibRes2 : CARDINAL;
BEGIN (* FibFct4; dient nur zur Anpassung der Schnittstelle *)
  FibProc (Arg, FibRes1, FibRes2); RETURN FibRes1;
END FibFct4;

```

2.3.4 Sprunganweisungen

MODULA-2 bietet im Gegensatz zu anderen Programmiersprachen zwar keine Sprunganweisung (i.a. GOTO-Befehl) an, doch soll wegen der grundsätzlichen Bedeutung auf die Problematik dieser Anweisung kurz eingegangen werden.

Sprünge in primitiven Sprachen

Assembler-Programm zur Berechnung der Fakultät:

```

LAT 1      {lade Adress-Teil = 1      }
SPA FAKUL  {Speichere als F im Akku  }
LDA ARGUM  {lade Argument  A        }
SKG ENDE   {Test auf A > 0           }
LOOP= MUL FAKUL {multipliziere  A * F }
SPA FAKUL  {Ergebnis ist neues F    }
LDA ARGUM  {lade erneut A           }
SAT 1      {subtrah.1 von Adress-Teil}
SPA ARGUM  {speichere wieder als A   }
SGR LOOP   {springen wenn A positiv  }
ENDE=...

```

In FORTRAN gibt es noch in enger Anlehnung an die obige Assembler-Notation das sogenannte *arithmetische IF*:

```
IF (N * 5 - 13) 112, 999, 5
```

d.h. wenn der Ausdruck $(N \cdot 5 - 13)$ null ergibt, wird das Programm bei *Marke* (Adresse) 999 fortgesetzt, bei einem negativen bzw. positiven Ergebnis bei Marke 112 bzw. Marke 5. Erst später wurde in FORTRAN auch das *logische IF* vorgesehen:

```
IF (N * 5 .GT. 13) GOTO 5
```

d.h. Sprung nach Marke 5, wenn $N \cdot 5$ größer ist als 13.

Sprünge in PASCAL

Ein Sprung in einem PASCAL-Programm erfordert:

eine *Sprunganweisung*, z.B. GOTO 72
 und die darin genannte Marke (Label) vor einer Anweisung 72 :

Diese *Marken-Definition* erfordert eine *Marken-Deklaration* im Kopf des betreffenden Programmabschnitts:

LABEL 72

Die Möglichkeit der Verwendung von Sprüngen ist durch viele Regeln eingeschränkt (beispielsweise darf man nicht *in* eine Prozedur oder *in* einen Zweig einer Alternative hineinspringen).

Beispiel (der folgende Programmausschnitt enthält zwei Fehler):

```

PROCEDURE P1;                                (* Programm P2.27 *)
  LABEL 333;
  FUNCTION F1 (U:real) : integer;
  BEGIN ...
    333: ...
  END {F1};
BEGIN {P1} ... GOTO 333; ...
END {P1};

```

Fehler 1: Label-Deklaration und -Definition befinden sich nicht auf der gleichen Ebene.

Fehler 2: Sprung in Struktur (F1) hinein.

Kritik am GOTO

E. Dijkstra hat mit seinem Brief "GOTO considered harmful" (1966) darauf aufmerksam gemacht, daß Programme durch Verwendung von Sprunganweisungen *undurchsichtig* und damit *fehlerträchtig* und kaum modifizierbar werden. Dies hat u.a. folgende Gründe: Jedes Verständnis des Programms oder seine *Verifikation* (der Nachweis, daß das Programm das tut, was es soll) setzen *Abstraktion* voraus.

Dazu ist es vorteilhaft (praktisch: Voraussetzung), daß Abstraktionen wie die Zusammenfassung von Befehlen in höheren Sprachkonstrukten gleiche strukturelle Eigenschaften wie eine einzelne Operation haben (*single Entry, single Exit*), d.h. eindeutige Einstiegs- und Ausgangsstellen in diesem Konstrukt. Diese Eigenschaft findet sich bei den in MODULA-2 verfügbaren Konstrukten wie FOR, WHILE, REPEAT, LOOP, IF, CASE.

Auch in Sprachen wie PASCAL, die den Sprung anbieten, gilt die Regel, daß man auf den bedingten Sprung verzichten und stattdessen den *kontrollierten Sprung* (die

Der zugeordnete Prozedurtyp ist dann

```
TYPE ProzTyp = PROCEDURE (REAL, VAR INTEGER, VAR INTEGER);
```

Man beachte, daß in der Definition von ProzTyp VAR wiederholt werden muß, sonst ist der letzte ein Wert-Parameter. Eine Prozedur FuehreAus kann nun deklariert werden mit

```
PROCEDURE FuehreAus (Arg: CHAR; F: ProzTyp);
```

Sie wird dann aufgerufen mit

```
FuehreAus ('X', TueIrgendEtwas);
```

Man verwendet Prozedur-Parameter dort, wo man einer (im Sinne der *Aufrufhierarchie*) untergeordneten Prozedur mitteilen will, welches spezielle Verfahren sie anwenden soll. Man kann sich z.B. ein Programm vorstellen, das den Bildschirm mit einem (variablen) regelmäßigen Muster beschreibt. Diesem Programm könnte man als Parameter eine Prozedur mitgeben, die ein Exemplar (also einen Ausschnitt) des Musters an bestimmter Stelle zeichnet.

Auch Referenz-Parameter sind erlaubt, so daß Prozeduren "von unten nach oben" gereicht und dort in *Prozedurvariablen* gespeichert werden können. Man gebrauche diese Möglichkeiten aber mit größter Zurückhaltung, denn die Programme werden dadurch sicher nicht verständlicher. Eine schwerwiegende Einschränkung bei der Verwendung der Prozedur-Parameter liegt in der Vorschrift, daß alle so übergebenen Prozeduren auf Modulebene definiert sein müssen.

Klassische Anwendungsbeispiele für Prozedurtypen sind *numerische* Algorithmen, z.B. zur Integration. Ein solcher Algorithmus für die Integralrechnung nach der Simpson-Methode ist nachfolgend angegeben (Programm P2.28). Für das Verständnis dieses Programms werden kurz die mathematischen Grundlagen erläutert.

Mathematische Grundlagen der Simpson-Integration

Die *Simpson-Integration* zählt zu den *numerischen* Integrationsmethoden, die durch einfache Berechnungen bestimmte Integrale möglichst genau zu approximieren versuchen. Die Simpson-Integration beruht auf der *Keplerschen Faßregel*. Diese Regel besagt, daß man eine gegebene Kurve i.a. besser durch einen Parabelbogen als durch ein Geradenstück approximieren kann.

Abb. 2.16 zeigt dies durch einen Parabelbogen mit den *Grenzpunkten* UG und OG sowie dem *Mittelpunkt* M, unter dem sich die schraffierte Fläche als Näherung des bestimmten Integrals

$\int_{UG}^{OG} f(x) dx$ ergibt.

Abb. 2.16 Keplersche Faßregel

Sei $p(x)$ die Parabelfunktion, die durch drei Punkte der zu integrierenden Funktion (bei UG, OG und M, also an den Grenzen und in der Mitte) bestimmt ist. Dann gilt:

$$\int_{UG}^{OG} p(x) dx \approx \int_{UG}^{OG} p(x) dx = \frac{OG - UG}{6} \cdot (f(UG) + 4 f(M) + f(OG)).$$

Verbessert man die Approximation durch Aufteilung des Intervalls, über dem integriert werden soll, in m gleichbreite Teilintervalle, so läßt sich die Berechnung auf jeden der m Streifen anwenden.

Sei $B = \frac{OG-UG}{2m}$ die halbe Breite jedes dieser m Streifen und

$P_1 = UG + B, P_2 = UG + 2B$ usw. bis $P_{2m-1} = OG - B$

die entstehenden Mittelpunkte (bei ungeradem Index) bzw. die neuen "inneren" Randpunkte (bei geradem Index). Dann ist die Summe aller Parabelintegrale

$$\begin{aligned} \int_{UG}^{OG} p^m(x) dx &= \frac{1}{6} \cdot 2 B \cdot (f(UG) + 4 f(P_1) + f(P_2) \\ &\quad + f(P_2) + 4 f(P_3) + f(P_4) \\ &\quad + \dots \\ &\quad + f(P_{2m-2}) + 4 f(P_{2m-1}) + f(OG)) \end{aligned}$$

$$\begin{aligned}
&= \frac{B}{3} \cdot (f(UG) + 4 f(P_1) + 2 f(P_2) + 4 f(P_3) + 2 f(P_4) \\
&\quad + \dots + 4 f(P_{2m-1}) + f(OG)) \\
&= \frac{B}{3} \cdot (\text{Grenzhöhen} + 2 \cdot \text{Randhöhen} + 4 \cdot \text{Mittelhöhen})
\end{aligned}$$

wobei die Grenzhöhen $f(UG) + f(OG)$,
 die Randhöhen $f(P_2) + f(P_4) + \dots + f(P_{2m-2})$ und
 die Mittelhöhen $f(P_1) + f(P_3) + \dots + f(P_{2m-1})$
 sind.

Die Idee des Algorithmus in P2.28 besteht darin, die Approximation schrittweise durch Halbierung der Intervallbreiten zu verbessern. Dabei werden die alten Mittelhöhen zu Randhöhen, so daß bei jeder Iteration zunächst die Mittelhöhen zu den Randhöhen addiert und dann die neuen Mittelhöhen berechnet werden. Dazu muß man nur die Summen, nicht jedoch die Summanden speichern.

Implementierung

```

MODULE Simpson;                                     (* Programm P2.28 *)
(* Integration frei nach dem Buch von DalCin, Lutz, Risse *)
FROM InOut      IMPORT (* PROC *) Write, WriteString, WriteLn;
FROM RealInOut  IMPORT (* PROC *) WriteReal;
FROM MathLib0   IMPORT (* PROC *) sin;
TYPE FunktionsTyp = PROCEDURE (REAL): REAL;

PROCEDURE Simpson (Funktion : FunktionsTyp;
                  Untergrenze, Obergrenze,
                  Toleranz : REAL) : REAL;
(* Integriert Funktion nach der Simpson-Regel von Unter- bis
   Obergrenze. Toleranz gibt die maximale relative Aenderung bei
   der Iteration vor (d.h. bei Halbierung der Streifenbreite). *)
VAR   index, Teile : INTEGER; (* Anzahl Streifen *)
      Breite: REAL; (* eigentlich halbe Breite eines Streifens *)
      Randhoehen, Mittelhoehen: REAL; (* Summe d. Streifenhoehen *)
      Grenzhoehen: REAL; (* Hoehen am Rand d. Integr. intervalls *)
      Summe, SummeAlt : REAL; (* zum Vergleich in Iteration *)
BEGIN (* Simpson *)
  (* Initialisierung *)
  Teile      := 1;
  Breite     := (Obergrenze - Untergrenze) / 2.0;
  Mittelhoehen := Funktion (Untergrenze + Breite);
  Randhoehen  := 0.0;
  Grenzhoehen := Funktion (Untergrenze) + Funktion (Obergrenze);
  Summe      := (Grenzhoehen + 4.0*Mittelhoehen) * Breite/3.0;

```

```

REPEAT
  (* Streifen aus der fruheren Rechnung werden halbiert, *)
  (* dadurch werden die Raender zur Mitte, *)
  (* Mittelhoehen werden neu berechnet *)
  SummeAlt      := Summe;
  Teile         := Teile * 2;
  Breite        := Breite / 2.0;
  Randhoehen    := Randhoehen + Mittelhoehen;
  Mittelhoehen  := 0.0;
  FOR index := 1 TO Teile DO
    Mittelhoehen := Mittelhoehen + Funktion (Untergrenze +
                                             Breite * FLOAT (2*index - 1));
  END (* FOR *);
  Summe := (Grenzhoehen + 4.0*Mittelhoehen + 2.0*Randhoehen) *
           Breite / 3.0;
UNTIL (ABS (Summe/SummeAlt - 1.0) <= Toleranz);
RETURN Summe;
END Simpson;

PROCEDURE Polynom (X: REAL) : REAL;
BEGIN (* Polynom *)
  RETURN X*X*X + 3.0 * X*X - 2.0 * X + 25.0;
END Polynom;

PROCEDURE Sinus2 (B: REAL) : REAL;
VAR H: REAL;
BEGIN (* Sinus2 *)
  H := sin (B); RETURN H*H;
END Sinus2;

CONST Diff = 1.0E-6;
      Pi   = 3.1415927;
BEGIN
  WriteString ("Polynom-Integral = :");
  WriteReal (Simpson (Polynom, -3.0, 3.0, Diff), 14);
  WriteLn;
  WriteString ("Sin**2 -Integral = :");
  WriteReal (Simpson (Sinus2, 0.0, 2.0 * Pi, Diff), 14);
  WriteLn;
END Simpson.

```

Das Programm erzeugt die folgende Ausgabe:

```

Polynom-Integral = :+2.0400000E+02
Sin**2 -Integral = :+3.1415927E+00

```

Im ersten Fall zeigt die Nachrechnung, daß 204 das korrekte Resultat ist. Die Simpson-Regel ist für Polynome bis zur dritten Potenz exakt. Auch im zweiten Fall ist das exakte Ergebnis π sehr genau erreicht.

2.4 Komplexe Datentypen

Unter komplexen Datentypen verstehen wir in diesem Kapitel Mengen/Sets (Abschnitt 2.4.1), Arrays/Felder (Abschnitt 2.4.2), Records/Verbunde (Abschnitt 2.4.3), "Zeigerstrukturen"/Pointer (Abschnitt 2.4.4) und Dateien/Files (Abschnitt 2.4.5).

2.4.1 Mengen (Sets)

In der Mathematik spielen Mengen (Sets) seit langem eine zentrale Rolle. Ihre Vorteile sind

- die Möglichkeit, beliebige und beliebig viele Gegenstände in Mengen zusammenzufassen
- die Möglichkeit, auch aus Mengen *rekursiv* wieder Mengen zu bilden
- die *Abstraktion* von einer bestimmten Reihenfolge, in der die Elemente der Menge notiert sind.

Diese Eigenschaften sind auch in der Programmierung sehr attraktiv. Doch leider sind es dieselben Eigenschaften, die eine Implementierung außerordentlich schwierig machen.

Darum tun sich traditionelle Programmiersprachen mit der Behandlung von Mengen schwer. Am weitesten ging PASCAL; bei MODULA-2 ist von diesen Möglichkeiten nur ein Rest geblieben. Die einzige (etwas) bekanntere Sprache, die die allgemeine Menge als *Grundelement* verwendet, ist SETL (Set-Language). Listen, also geordnete Mengen, sind das wesentliche Merkmal der Sprache LISP (List Processing Language).

In PASCAL und MODULA-2 sind die *Sets* nur magere Imitationen mathematischer Mengen, da sie jeweils nur über einen bestimmten primitiven Basistyp mit sehr begrenztem Wertevorrat gebildet werden können. Die Größe ist implementierungsabhängig, aber in PASCAL i.a. groß genug, um die Menge der Zeichen darzustellen; in MODULA-2 ist die Grenze zugunsten einer sehr effizienten Realisierung tiefer gelegt, so daß der Basistyp auf die Wortbreite des Prozessors, i.a. 16 oder 32 Werte beschränkt ist.

2.4.1.1 Darstellung und Manipulation von Mengen

Wenn wir einen Datentyp verwenden, der eine Menge repräsentiert, so benötigen wir die nachfolgend vorgestellten Notationen und Operationen. Mengentypen werden eingeführt durch

```
TYPE ST = SET OF BT;
```

Dies definiert einen *Mengentyp* ST über einem *Basistyp* BT. BT muß ein Unterbereichs- oder Aufzählungstyp sein, der höchstens N Elemente hat. N ist systemabhängig, typisch 16 oder 32:

```
ORD (MAX (BT)) - ORD (MIN (BT)) < N
```

Handelt es sich beim Basistyp um einen Unterbereich der Typen INTEGER, CARDINAL oder CHAR, so gibt es meist eine weitere wichtige Implementierungsrestriktion:

```
ORD (MAX (BT)) < N
```

Dadurch ist es möglich, die Werte ohne Verschiebung (also sehr effizient) zu verarbeiten.

Beispiel:

```
TYPE Farben      = (Rot, Gelb, Gruen, Blau, Schwarz, Weiss);
   Farbmenge = SET OF Farben;
```

Nun können wir Variablen des Typs ST definieren und damit arbeiten. Syntax in MODULA-2:

```
ST{ }           leere Menge des Typs ST
ST{c1, ..., cn} Mengenkonstante des Mengentyps ST;
                c1 bis cn müssen Konstanten des Basistyps sein.
```

Die Symbole der Grundrechenarten und einiger Relationen sind auch auf Paare *typgleicher* Mengen anwendbar, doch haben sie dabei eine mengentheoretische Bedeutung:

S1 + S2	<i>Vereinigung</i>	der Mengen S1 und S2
S1 * S2	<i>Schnitt</i>	der Mengen S1 und S2
S1 - S2	<i>Differenz</i>	der Mengen S1 und S2
S1 / S2	<i>Symmetrische Differenz</i>	der Mengen S1 und S2
S1 <= S2	S1 ist <i>Untermenge</i> von S2	
S1 >= S2	S1 ist <i>Obermenge</i> von S2	

Die *Aufnahme* (Inklusion) und der *Ausschluß* (Exklusion) von Elementen geschieht mit INCL und EXCL:

$$\text{INCL}(S, V) \equiv S + \{V\}$$

$$\text{EXCL}(S, V) \equiv S - \{V\}$$

Der *Test*, ob ein Element in der Menge ist, geschieht mit dem Operator IN:

$$V \text{ IN } S \equiv V \in S$$

Mit IN können die Operationen und Relationen wie folgt definiert werden:

$$x \text{ IN } (S1 + S2) \equiv (x \text{ IN } S1) \quad \text{OR} \quad (x \text{ IN } S2)$$

$$x \text{ IN } (S1 - S2) \equiv (x \text{ IN } S1) \quad \text{AND} \quad \text{NOT } (x \text{ IN } S2)$$

$$x \text{ IN } (S1 * S2) \equiv (x \text{ IN } S1) \quad \text{AND} \quad (x \text{ IN } S2)$$

$$x \text{ IN } (S1 / S2) \equiv (x \text{ IN } S1) \quad \neq \quad (x \text{ IN } S2)$$

$$S1 \leq S2 \equiv \forall x: (x \text{ IN } S1) \Rightarrow (x \text{ IN } S2)$$

$$S1 \geq S2 \equiv \forall x: (x \text{ IN } S2) \Rightarrow (x \text{ IN } S1)$$

In MODULA-2 gibt es einen *vordefinierten* Set-Typ, den Typ BITSET. Er ist systemabhängig definiert als

TYPE BITSET = SET OF [0..N-1]

worin N i.a. die Wortlänge des Rechners ist. Syntaktisch gibt es zu den allgemeinen Sets nur den Unterschied, daß Bitset-Konstanten ohne Typangabe geschrieben werden dürfen. Alle anderen Sets werden *intern* als BITSET behandelt.

Variablen des Typs BITSET dienen in der Praxis nicht zur Beschreibung ungeordneter Mengen, sondern sie repräsentieren *Bitmuster*, wie sie in der *Systemprogrammierung* eine Rolle spielen. Mit den Mengenoperationen können dann Teile der Bitmuster ausgeblendet, gesetzt oder gelöscht werden.

2.4.1.2 Ein Beispiel für Sets

Das Programm P2.29 demonstriert einige Anwendungen der Mengen-Operationen.

```

MODULE Sets;                                     (* Programm P2.29 *)
(* Demo der Set-Operationen *)
FROM InOut IMPORT (* PROC *) Write, WriteString,
                  WriteInt, WriteLn;
TYPE FarbTyp      = (Rot, Orange, Gelb, Gruen, Blau, Violett);
   FarbSetTyp     = SET OF FarbTyp;

```

```

PROCEDURE ShowSet (EinSet: FarbSetTyp; Text: ARRAY OF CHAR);
VAR i : FarbTyp;
BEGIN (* ShowSet *)
  FOR i := MIN (FarbTyp) TO MAX (FarbTyp) DO
    IF i IN EinSet THEN Write('X') ELSE Write('-') END;
  END (* FOR *);
  WriteString (' '); WriteString (Text); WriteLn;
END ShowSet;

VAR Farbe          : FarbTyp;
    Farben1, Farben2 : FarbSetTyp;

PROCEDURE Vergleich;
(* Verwendet globale Variablen Farben1 und Farben2 lesend *)
  PROCEDURE WriteNot (Fall: BOOLEAN; Text: ARRAY OF CHAR);
  BEGIN (* WriteNot *)
    WriteString (' ');
    IF Fall THEN WriteString (' ');
      ELSE WriteString ('NOT ');
    END (* IF *);
    WriteString (Text); WriteLn;
  END WriteNot;
BEGIN (* Vergleich *)
  WriteNot (Farben1 <= Farben2, 'F1 <= F2');
  WriteNot (Farben1 >= Farben2, 'F1 >= F2');
  WriteNot (Farben1 = Farben2, 'F1 = F2');
  WriteLn;
END Vergleich;

CONST SetLeer = FarbSetTyp{}; (* Mengenkonstanten *)
    SetVoll = FarbSetTyp{Rot,Orange,Gelb,Gruen,Blau,Violett};
BEGIN (* Sets *)
  Farben1 := SetLeer; ShowSet (Farben1, 'F1');
  Farben2 := SetVoll - FarbSetTyp{Blau}; (* Differenz *)
    ShowSet (Farben2, 'F2');
  Farben2 := Farben2 * FarbSetTyp{Blau,Rot,Violett};
    (* Schnitt *)
    ShowSet (Farben2, 'F2 nach *');

  Vergleich;
  INCL (Farben1,Gelb); (* 2. Aufnahme von Gelb *)
  INCL (Farben1,Gelb); (* ist natuerlich wirkungslos *)
  INCL (Farben1,Gruen); ShowSet (Farben1, 'F1 nach INCL');
  Farben2 := Farben2 + Farben1; (* Vereinigung *)
    ShowSet (Farben2, 'F2 nach +');
  Farben2 := Farben2 - FarbSetTyp {Gelb, Blau, Rot};
    (* Differ. *)
    ShowSet (Farben2, 'F2 nach -');

  Vergleich;
  (* Trickreiches Vertauschen durch dreimaliges exkl. ODER *)
    ShowSet (Farben1, 'F1 vor Tausch');
    ShowSet (Farben2, 'F2 vor Tausch');
  Farben1 := Farben1 / Farben2; ShowSet (Farben1, 'F1 nach /');
  Farben2 := Farben1 / Farben2; ShowSet (Farben2, 'F2 nach /');

```

```

Farben1 := Farben1 / Farben2; ShowSet (Farben1, 'F1 nach /');
WriteLn;
Farben1 := Farben2;          ShowSet (Farben1, 'nach F1 := F2');
Vergleich;
END Sets.

```

Die Ausgabe des Programms Sets sieht so aus:

```

-----      F1
XXXX-X      F2
X----X      F2 nach *
                        F1 <= F2
                        NOT F1 >= F2
                        NOT F1 = F2

--XX--      F1 nach INCL
X-XX-X      F2 nach +
---X-X      F2 nach -
                        NOT F1 <= F2
                        NOT F1 >= F2
                        NOT F1 = F2

--XX--      F1 vor Tausch
---X-X      F2 vor Tausch
--X--X      F1 nach /
--XX--      F2 nach /
---X-X      F1 nach /
--XX--      nach F1 := F2
                        F1 <= F2
                        F1 >= F2
                        F1 = F2

```

2.4.2 Arrays (Felder)

Arrays (Felder) sind *homogene* kartesische Produkte und dienen zur Aufnahme mehrerer Informationen des gleichen Typs. Auf die Elemente eines Arrays kann über einen *Index* bzw. mehrere *Indizes* (im Fall mehrdimensionaler Arrays) in eckigen Klammern zugegriffen werden. Zur Indizierung kann jeder skalare Typ (außer REAL) verwendet werden. Dem Array als Datenstruktur entspricht die *Laufschleife* (siehe Abschnitt 2.3.3) als Ablaufstruktur. Bei mehrfach indizierten Variablen gibt es entsprechend geschachtelte Schleifen.

Beispiel (Fragment):

```

FROM InOut IMPORT (* PROC *) Read;
CONST FeldLng      = 100;
      NamensLng     = 15;
TYPE  FeldNrTyp    = [1..FeldLng];
      FeldIdxTyp    = [0..FeldLng];

```

(* Programm P2.30 *)


```

        NamensIdxTyp = [1..NamensLng];
        NamensTyp    = ARRAY NamensIdxT OF CHAR;
VAR   Name          : NamensTyp;
      Feld          : ARRAY FeldNrTyp OF NamensTyp;
      FeldIdx       : FeldIdxTyp;
      ZeichenIdx    : NamensIdxTyp;
      Zeichen       : CHAR;

BEGIN (* P2.30 *)
  REPEAT
    FOR ZeichenIdx := 1 TO NamensLng DO
      Read (Name [ZeichenIdx]);
    END;
    FeldIdx := 0;
    IF Name [1] # " " THEN          (* Abbruch falls Name mit
                                   Leerzeichen beginnt *)
      INC (FeldIdx); Feld [FeldIdx] := Name;
    END;
  UNTIL FeldIdx = FeldLng;
END P2.30;

```

Der Indexbereich eines Arrays sollte wie in diesem Beispiel als Datentyp definiert werden. Für die Deklaration der Indexvariablen wird oft noch ein weiterer Typ, der um 1 "breiter" ist, benötigt.

Man beachte, daß der Index eines Arrays nicht mit 0 oder 1 beginnen muß, er kann auch im negativen Bereich liegen oder einen Aufzählungstyp haben. Z.B. ist BOOLEAN gelegentlich als Indexbereich sehr sinnvoll, weil man dann einen boolschen Ausdruck als Index verwenden kann.

Die Elemente eines Arrays können von beliebigem Typ, also auch wieder Arrays sein. In diesem Falle ergibt sich bei der Deklaration eine *Schachtelung* der Art

```

ARRAY IndexTyp1 OF
  ARRAY IndexTyp2 OF
    ... OF ARRAY IndexTypn OF BasisTyp;

```

Eine solche Deklaration läßt sich auch zusammenfassen durch

```

ARRAY IndexTyp1, IndexTyp2, ..., IndexTypn OF BasisTyp;

```

Zwischen ganzen Arrays ist (bei exakter Typgleichheit) die *Wertzuweisung* erlaubt. Vergleiche (z.B. mit "=", ">") sind dagegen nicht zulässig. Exakte Typgleichheit besteht nur, wenn zwei Felder mit demselben Typ vereinbart sind oder bei verschiedenen Typen, wenn der eine durch den anderen definiert ist.

Beispiel:

```

(* Programm P2.31 *)
TYPE FT1      = ARRAY [-3..+3] OF REAL;
   FT2      = FT1;
VAR  F0, F1 : FT1;
     F2      : FT2;
     F3, F4 : ARRAY [-3..+3] OF REAL;
     F5      : ARRAY [-3..+3] OF REAL;

```

In diesem Falle sind die Variablen F0, F1 und F2 typgleich, ebenso F3 und F4, dagegen F1, F3 und F5 nicht !

Strings

I.a. können in MODULA-2 keine Array-Konstanten definiert werden. Eine Ausnahme bilden Konstanten des Typs `ARRAY [0..n-1] OF CHAR`, also *Zeichenreihen* oder *Strings*. Diese dürfen nach dem für Konstanten üblichen Schema definiert werden, beispielsweise

```
CONST Anfangsmeldung = 'Start des Dialogprogramms';
```

Solche String-Konstanten gehören aber in Wirklichkeit zu einem speziellen Typ, der mit Variablen des Typs `ARRAY [0..m-1] OF CHAR` zuweisungskompatibel ist (falls $m \geq n$). Bei einer Zuweisung mit $m > n$ wird ein Schlußzeichen angehängt, das das Ende des Strings markiert (üblicherweise das Zeichen 0C).

Dagegen kann man auf einzelne Zeichen in der String-Konstanten nicht direkt zugreifen, etwa mit

```
Write (Anfangsmeldung [0]);    (* FEHLER *)
```

Wird also ein *zeichenweiser Zugriff* gewünscht, so muß eine Zuweisung oder eine Übergabe als Parameter vorgenommen werden. In MODULA-2 gibt es über diese String-Konstanten hinaus *keine* Zeichenreihenverarbeitung (*String Handling*) wie in manchen anderen Sprachen. Allerdings kann diese vom Programmierer realisiert werden.

Ein wesentliches Merkmal jeder Zeichenreihe ist ihre effektive *Länge*, also die Zahl der darin enthaltenen Zeichen. MODULA-2 legt nicht fest, wie diese Länge gespeichert wird; die beiden gängigen Möglichkeiten sind die *explizite* Speicherung der Länge und die *Markierung* des Endes durch ein spezielles Schlußzeichen (0C). Dieses entfällt nur dann, wenn eine Zuweisung an eine genau passende String-Variable erfolgt.

Beispiel:

```

VAR   StringVar : ARRAY [0..9] OF CHAR;    (* Programm P2.32 *)
CONST StringConst = '0123456';
BEGIN (* P2.32 *)
  StringVar := '0123456789'; (* voll genutzt, kein Schlusssz. *)
  StringVar := StringConst;  (* Schlusssz. an Stelle der '7' *)
  Write(StringConst [3]);    (* ist falsch ! *)
  Write(StringVar [3]);      (* korrekt, liefert '3' *)
END P2.32;

```

Die unterschiedliche Behandlung von Strings führt häufig zu Problemen bei der Übertragung (Portierung) von MODULA-2-Programmen auf Rechner mit anderen Übersetzern.

Flexible Array-Grenzen bei Parametern

Der Zwang, den Typ von Arrays zur *Übersetzungszeit* festzulegen, ist besonders bei Prozeduren hinderlich. Hier hilft das Prinzip der *flexiblen* Array-Grenzen. Auf diese Weise kann dieselbe Prozedur für (eindimensionale) Arrays verschiedener Länge verwendet werden. Allerdings muß der Basistyp gleich sein. Der Indexbereich ist in der Prozedur abgebildet auf den Bereich von 0 bis N-1, wenn N die Zahl der Indexelemente ist, unabhängig von den absoluten Indexgrenzen des konkreten Parameters. Tatsächlich kann dieser sogar durch einen Aufzählungstyp indiziert sein. Der aktuelle Wert von N-1 kann durch Aufruf der Funktion HIGH bestimmt werden.

Beispiel:

```

(* Programm P2.33 *)
PROCEDURE TestFlex (VAR IntFeld : ARRAY OF INTEGER;
                   StringPar: ARRAY OF CHAR);
VAR i, Summe : INTEGER;
BEGIN (* TestFlex *)
  Summe := 0;
  FOR i := 0 TO HIGH (IntFeld) DO
    Summe := Summe + IntFeld [i];
  END (* FOR *);
  FOR i := 0 TO HIGH (StringPar) DO
    Write (StringPar [i]);
  END;
END TestFlex;

```

Man beachte, daß die Syntax "ARRAY OF BT" keinen neuen Typ definiert, sondern *nur* in Parameterlisten zugelassen ist (während die Konstruktion von Typen dort ausgeschlossen ist). Es gibt also in MODULA-2 keine Möglichkeit, Felder direkt als flexibel zu *deklarieren* (beispielsweise im Hauptprogramm)!

Beispiel für Arrays: die Matrix-Multiplikation

Programm P2.34 zeigt die Multiplikation zweier Matrizen. Es ist, um möglichst viele spezielle Fälle zu zeigen, erheblich umfangreicher als nötig.

```

MODULE Arrays;                                     (* Programm P2.34 *)
(* Zur Demonstration der Arrays *)
FROM InOut IMPORT (* PROC *) WriteInt, WriteString, WriteLn;
CONST L1 = 6; L3 = 5;
TYPE IndBer1 = INTEGER[1..L1];    (* nicht CARDINAL-Subrange *)
IndBer2 = (Alpha, Beta, Gamma);
    IndBer3 = INTEGER[1..L3];
    Vektor = ARRAY IndBer3 OF INTEGER;
    Matrix1 = ARRAY IndBer1, [Alpha..Gamma] OF INTEGER;
    Matrix2 = ARRAY      [Alpha..Gamma] OF Vektor;
    Matrix3 = ARRAY IndBer1      OF Vektor;
VAR M1 : Matrix1; M2 : Matrix2; M3 : Matrix3;
(* werden in den Prozeduren als globale Variablen bearbeitet *)

PROCEDURE DruckZeile (Zeile: ARRAY OF INTEGER);
(* Ausgabe einer Zeile einer (beliebig breiten) *)
(* Integer-Matrix *)
VAR i : INTEGER;
BEGIN (* DruckZeile *)
    FOR i := 0 TO HIGH (Zeile) DO WriteInt (Zeile [i], 13) END;
    WriteLn;
END DruckZeile;

PROCEDURE Vorbesetzung;    (* Vorbesetzung von M1 und M2 *)
(* einfache Vorbesetzung der Matrizen mit laufenden Werten *)
VAR i : IndBer1; j : IndBer2; k : IndBer3; h : INTEGER;
BEGIN (* Vorbesetzung *)
    h := 1;
    FOR i := 1 TO L1 DO
        FOR j := MIN(IndBer2) TO MAX(IndBer2) DO
            M1 [i, j] := h; INC (h);
        END (* FOR j *);
    END (* FOR i *);
    h := -6;
    FOR j := MIN(IndBer2) TO MAX(IndBer2) DO
        FOR k := 1 TO L3 DO
            M2 [j, k] := h; INC (h);
        END (* FOR k *);
    END (* FOR j *);
END Vorbesetzung;

PROCEDURE Ausgabel;    (* Ausgabe M1 und M2 *)
VAR i : IndBer1; j : IndBer2;
BEGIN (* Ausgabel *)
    WriteLn; WriteString ('Matrix M1'); WriteLn;
    FOR i := 1 TO L1 DO DruckZeile (M1 [i]) END;

```

```

WriteLn; WriteLn; WriteString ('Matrix M2'); WriteLn;
FOR j := MIN(IndBer2) TO MAX(IndBer2) DO
  DruckZeile (M2[j]);
END (* FOR j *);
WriteLn;
END Ausgabel;

PROCEDURE Multiplikation;
(* Matrixmultiplikation M3 := M1 * M2 *)
VAR i : IndBer1; j : IndBer2; k : IndBer3;
    Summe : INTEGER;
BEGIN (* Multiplikation *)
  FOR i := 1 TO L1 DO
    FOR k := 1 TO L3 DO
      Summe := 0;
      FOR j := MIN(IndBer2) TO MAX(IndBer2) DO
        Summe := Summe + M1 [i, j] * M2 [j, k];
      END (* FOR j *);
      M3 [i, k] := Summe;
    END (* FOR k *);
  END (* FOR i *);
END Multiplikation;

PROCEDURE Ausgabe2;      (* Ausgabe M3 *)
VAR i : IndBer1;
BEGIN (* Ausgabe2 *)
  WriteLn; WriteString ('Matrix M3'); WriteLn;
  FOR i := 1 TO L1 DO DruckZeile (M3 [i]) END; WriteLn;
END Ausgabe2;

PROCEDURE Drehen;      (* Zeilenvertauschung in M3 *)
VAR i : IndBer1; Mhilf : Matrix3;
BEGIN (* Drehen *)
  FOR i := 1 TO L1 DO
    Mhilf [L1+1-i] := M3 [i];    (* Zuweisung von Zeilen *)
  END (* FOR *);
  M3 := Mhilf;                  (* ganze Matrix wird zugewiesen *)
END Drehen;

BEGIN (* Arrays *)
  Vorbesetzung;      Ausgabel;
  Multiplikation;     Ausgabe2;
  Drehen;            Ausgabe2;
END Arrays.

```

Ausgabe (des Programms Arrays):

```

Matrix M1
      1      2      3
      4      5      6
      7      8      9
     10     11     12
     13     14     15
     16     17     18

```

Matrix M2	-6	-5	-4	-3	-2
	-1	0	1	2	3
	4	5	6	7	8
Matrix M3	4	10	16	22	28
	-5	10	25	40	55
	-14	10	34	58	82
	-23	10	43	76	109
	-32	10	52	94	136
	-41	10	61	112	163
Matrix M3	-41	10	61	112	163
	-32	10	52	94	136
	-23	10	43	76	109
	-14	10	34	58	82
	-5	10	25	40	55
	4	10	16	22	28

2.4.3 Records (Verbunde)

Records (Verbunde) sind *heterogene* kartesische Produkte und dienen zur Darstellung *inhomogener*, aber zusammengehöriger Informationen. Typische Beispiele sind

- Personendaten (bestehend aus Name, Adresse, Jahrgang, Geschlecht)
- Meßwerte (Zeit, Gerät, Wert)
- Strings (tatsächliche Länge, Inhalt)

und alle anderen komplexen, strukturierten Informationen über irgendwelche Objekte. Records repräsentieren also inhomogene, Arrays hingegen homogene kartesische Produkte. In den meisten Fällen werden die Records als Bausteine größerer Strukturen, wie z.B. Arrays, verwendet.

2.4.3.1 Einfache Records

Das Deklarationsschema für Records (vgl. P2.35) ist ganz ähnlich wie bei der formalen Parameterliste; jede Komponente erhält einen Record-internen Namen, den *Selektor*. Mehrere Selektoren für Felder gleichen Typs können auch zusammengefaßt werden (wie in der Parameterliste). Die Selektoren müssen (nur) innerhalb des speziellen Records eindeutig sein:

```

TYPE RecTyp = RECORD                                     (* Programm P2.35 *)
    Selektor1 : Typ1;
    ...
    Sel1, Selj: Typij;
    ...
    Selektorn : Typn;
END (* RECORD *);

```


Der Zugriff auf eine Komponente des Records ist möglich durch den Namen des Records, an den ein Punkt und der Selektor angehängt sind:

```
Rec1.Sel1
```

Natürlich kann das mit allen anderen Selektionsarten (z.B. *Indizierung*) kombiniert sein, so daß recht lange Variablen-Namen entstehen.

Beispiel: Das folgende Fragment zeigt einen Record zur Speicherung von Personendaten.

```
TYPE NamString = ARRAY [1..15] OF CHAR;    (* Programm P2.36 *)
   AnredeT     = (Herr, Frau);
   NameRec     = RECORD
       Vorname, Nachname : NamString;
       Anrede           : AnredeT;
   END (* RECORD *);
   PersonRec = RECORD
       Name : NameRec
       ... (* weitere Felder des Records *)
   END (* RECORD *);
VAR Personal: ARRAY [1..n] OF PersonRec;
```

Der Zugriff auf den ersten Buchstaben des Vornamens erfolgt nun mit

```
Personal [Index].Name.Vorname [1]
```

Die WITH-Anweisung

Die oben gezeigte Notation kann zu sehr schwerfälligen Programmen führen, vor allem, wenn *viele* Zugriffe auf Komponenten von Record-Variablen zusammenkommen. Für diesen Fall gibt es eine notationelle *Vereinfachung*: WITH und der Name eines Records gefolgt von DO werden einer Anweisungssequenz (Abschnitt), die mit END abgeschlossen ist, vorangestellt; in einem solchen Abschnitt kann der Name des Records (und der Punkt) dann weggelassen werden.

Beispiel: Das Programm enthalte die folgende Anweisung

```
                                     (* Programm P2.37a *)
IF ('a' <= Personal [Index].Name.Vorname [1]) AND
   (Personal[Index].Name.Vorname [1] <= 'z')
THEN
   Personal[Index].Name.Vorname [1] := CHR (ORD ('A') -
       ORD ('a') + ORD (Personal [Index].Name.Vorname {1}));
END (* IF *);
```

Mit Hilfe der WITH-Anweisung kann das auch geschrieben werden als

```
WITH Personal [Index] DO                                (* Programm P2.37b *)
  IF ('a' <= Name.Vorname [1]) AND (Name.Vorname [1] <= 'z')
  THEN
    Name.Vorname [1] := CHR (ORD ('A') - ORD ('a') +
                           ORD (Name.Vorname [1]));
  END (* IF *);
END (* WITH *);
```

Ebenso ist korrekt:

```
WITH Personal [Index].Name DO                            (* Programm P2.37c *)
  IF ('a' <= Vorname [1]) AND (Vorname [1] <= 'z')
  THEN
    Vorname [1] := CHR (ORD ('A') - ORD ('a') +
                       ORD (Vorname [1]));
  END (* IF *);
END (* WITH *);
```

Die erste Zeile kann ebenfalls in der folgenden Form geschrieben werden:

```
WITH Personal [Index] DO WITH Name DO ... END; END;
```

Es ist unzulässig, im Anweisungsteil einer WITH-Anweisung implizit einen *anderen* Record zu wählen (etwa im Beispiel oben durch Änderung von Index). Durch WITH-Anweisungen entstehen *Gültigkeitsbereiche* für die betreffenden Selektoren. Dadurch können gleiche Bezeichner oder Selektoren mit anderer Bedeutung verdeckt werden. Maßgeblich ist auch hier die statische Schachtelung der Anweisungen.

Das folgende Beispiel zeigt Records, auch als *geschachtelte* Records, und die Verwendung der WITH-Anweisung in verschiedenen Varianten:

```
MODULE RecTest;                                          (* Programm P2.38 *)
(* Demonstration der Records *)
FROM InOut IMPORT (* PROC *) WriteString, WriteCard, WriteLn;
CONST MaxPers = 10; MaxLng = 30;
TYPE DatumT = RECORD
  Jahr   : [0..3000];
  Monat  : [1..12];
  Tag    : [1..31];
END (* RECORD *);
String = ARRAY [0..MaxLng] OF CHAR;
NamTyp = RECORD (* mit drei Feldern gleichen Typs *)
  VorName, NachName, GebName : String;
END (* RECORD *);
GebAng = RECORD (* bestehend aus Record und Array *)
  Datum: DatumT;
  Ort:   String;
END (* RECORD *);
```

```

EheAng  = RECORD
    Stand : (ledig, verh, verw, gesch, gest);
    seit  : DatumT;
    (* Partner : PersonenAng;  waere falsch ! *)
END (* RECORD *);
PersonenAng = RECORD
    Name      : NamTyp;
    Geburt    : GebAng;
    ZivStand  : EheAng;
END (* RECORD *);
VAR Person1, Person2 : PersonenAng;
    Datum1, Datum2   : DatumT;
    Klasse           : ARRAY [1..MaxPers] OF PersonenAng;
    Name             : NamTyp;

BEGIN (* RecTest *)
    Person1.Name.VorName := 'Albert';
    Person1.Name.NachName := 'Einstein'; (* sehr umstaendlich! *)
    WITH Datum1 DO Jahr := 1879; Monat:= 3; Tag := 14; END;
    WITH Person1 DO
        WITH ZivStand DO (* das aeussere WITH bleibt wirksam *)
            Stand      := gest;
            seit        := Datum1;
            seit.Jahr   := 1955;
        END (* WITH ZivStand *);
        Geburt.Datum := Datum1; (* Wertzuw. eines Teil-Records *)
        Geburt.Ort   := 'Ulm';
    END (* WITH Person1 *);
    Person2 := Person1;          (* Wertzuweisung eines Records *)
    Klasse [5] := Person1;
    WITH Klasse [3] DO
        Name := Person2.Name;    (* d.h. Klasse[3].Name :=
                                   Person2.Name *)
    END (* WITH Klasse [3] *);
END RecTest.

```

Wie das folgende (unschöne) Fragment zeigt, dürfen WITH-Anweisungen *geschachtelt* werden. Alle Selektoren bleiben sichtbar bis zum Ende des WITH-Anweisung. Innen können sie nur durch gleichnamige Selektoren *verdeckt* werden.

```

(* Programm P2.39a *)
WITH Person1 DO (* oeffnet Name, Geburt, ZivStand *)
    WITH ZivStand DO (* oeffnet Stand, seit *)
        Stand := gest; (* Person1.ZivStand.Stand *)
        seit := Datum1; (* Person1.ZivStand.seit *)
        seit.Jahr := 1955; (* Person1.ZivStand.seit.Jahr *)
        Geburt.Datum := Datum1; (* Person1.Geburt.Datum *)
        Geburt.Ort := 'Ulm'; (* Person1.Geburt.Ort *)
    END (* WITH ZivStand *);
END (* WITH Person1 *);

```


Damit ist auch das Folgende zulässig (ohne schöner zu sein):

```

(* Programm P2.39b *)
WITH Person1 DO
  WITH ZivStand DO
    WITH Geburt DO
      Stand := gest; (* Person1.ZivStand.Stand *)
      seit := Datum1; (* Person1.ZivStand.seit *)
      seit.Jahr := 1955; (* Person1.ZivStand.seit.Jahr *)
      Datum := Datum1; (* Person1.Geburt.Datum *)
      Ort := 'Ulm'; (* Person1.Geburt.Ort *)
    END (* WITH Geburt *);
  END (* WITH ZivStand *);
END (* WITH Person1 *);

```

Dagegen können nicht mit *einer* WITH-Anweisung Selektoren *mehrerer* Ebenen sichtbar werden.

```

(* Programm P2.39c *)
WITH Person1.ZivStand DO
  Stand := gest;
  seit := Datum1;
  seit.Jahr := 1955;
  Geburt.Datum := Datum1; (* FEHLER *)
  Geburt.Ort := 'Ulm'; (* FEHLER *)
END (* WITH *);

```

Records können auch direkt geschachtelt definiert werden. Ein Teil-Record stellt jedoch keinen eigenen Typ dar und kann deshalb nicht unabhängig vom umgebenden Record verwendet werden (z.B. als Parameter eines Prozeduraufrufs).

```

(* Programm P2.39d *)
TYPE PersonenAng = RECORD (* direkt geschachtelte Definition *)
  Name : NamTyp;
  Geburt : RECORD
    Datum: DatumT;
    Ort: String;
  END (* RECORD Geburt *);
  ZivStand: EheAng;
END (* RECORD *);

```

Bei der Arbeit mit Records gilt für einen guten *Programmierstil*:

- WITH-Anweisungen verwenden,
- dabei hohe *Lokalität* anstreben.

Beispiel für Records

Das fragmentarische Programm P2.40 realisiert einen Bibliothekskatalog durch eine Tabelle mit Records.

```

MODULE Bibliothek;                                     (* Programm P2.40 *)
CONST MaxBuecher   = 100;
      LngAutor     = 20; (* Anzahl Zeichen im Namen *)
      LngTitel     = 50; (* Anzahl Zeichen im Titel *)
      MinLeserID   = 1000;
      MaxLeserID   = 9999; (* soll vierstellig sein *)
      MaxBesteller = 3;
TYPE BuchIndexT   = [1..MaxBuecher];
      BuchZahlT   = [0..MaxBuecher];
      AutorIndT   = [1..LngAutor];
      TitelIndT   = [1..LngTitel];
      BestellIndT = [1..MaxBesteller];
      BestZahlT   = [0..MaxBesteller];
      SignT       = CARDINAL;
      StatusT     = (bestellt, vorhanden, ausgeliehen,
                     nichtverfuegbar, vermisst);
      LeserIDT    = [MinLeserID..MaxLeserID];
      AutorT      = ARRAY AutorIndT OF CHAR;
      TitelT      = ARRAY TitelIndT OF CHAR;
      BestListenT = ARRAY BestellIndT OF LeserIDT;
      BuchT       = RECORD
                          Autor       : AutorT;
                          Titel      : TitelT;
                          Signatur   : SignT;
                          ZahlBesteller: BestZahlT;
                          Besteller  : BestListenT;
                          Status     : StatusT;
                          Ausleiher  : LeserIDT;
                          Interessent : LeserIDT;
                        END (* RECORD *);
VAR Katalog       : ARRAY BuchIndexT OF BuchT;
                  (* global verwendet *)
      ZahlBuecher  : BuchZahlT; (* global verwendet *)
      Erfolg      : BOOLEAN;

PROCEDURE Bestellung (  Autor1: AutorT;
                       Titell: TitelT;
                       Signl  : SignT;
                       VAR OK   : BOOLEAN);
(* bestelltes Buch wird in den Katalog aufgenommen *)
(* OK = FALSE, wenn Katalog voll oder Signl bereits verwendet
   Katalog bleibt in diesem Falle unveraendert *)
VAR Index: BuchZahlT;
BEGIN (* Bestellung *)
  OK := (ZahlBuecher < MaxBuecher);
  IF OK THEN (* Buch provisorisch eintragen *)
    WITH Katalog [ZahlBuecher + 1] DO
      Autor       := Autor1;
      Titel      := Titell;
      Signatur    := Signl;
      ZahlBesteller := 0;
      Status     := bestellt;
    END (* WITH *);
  END (* IF *);
END (* BEGIN *);

```

```

    (* Pruefung, ob die Signatur bereits vorhanden war *)
    Der neue Eintrag wirkt als Ende-Markierung (Sentinel) *)
    Index:= 0;
    REPEAT INC (Index) UNTIL Katalog[Index].Signatur = Sign1;
    OK := Index > ZahlBuecher;  (* TRUE, wenn Neueintrag
                                gefunden *)
    IF OK THEN INC (ZahlBuecher)  END; (* Eintrag fixieren *)
    END (* IF OK *);
END Bestellung;

PROCEDURE Loeschen (LoeschSign: SignT; VAR OK: BOOLEAN);
(* OK = FALSE, wenn LoeschSign nicht gefunden wurde *)
VAR Index : BuchZahlT;
BEGIN (* Loeschen *)
    (* Eintrag suchen *)
    OK:= FALSE;  Index:= 0;
    WHILE (Index < ZahlBuecher) AND NOT OK DO
        INC (Index); OK:= (Katalog[Index].Signatur = LoeschSign);
    END (* WHILE *);
    IF OK THEN (* letzter Eintr. an d. Stelle d. geloeschten *)
        Katalog [Index]:= Katalog [ZahlBuecher];
        DEC (ZahlBuecher);
    END (* IF *); (* funktion. auch, w. Index = ZahlBuecher *)
END Loeschen;

(*es folgen weitere Prozeduren ... *)

BEGIN (* Bibliothek *)
    (*... Hauptprogramm ...*)
END Bibliothek.

```

2.4.3.2 Records mit Varianten

In der Praxis ist es oft zweckmäßig und natürlich, zwei Typen einfach als *Varianten* des *gleichen* Typs zu betrachten. Zum Beispiel kann ein Typ Koordinaten als Vereinigung der beiden Varianten der kartesischen und polaren Koordinaten betrachtet werden, deren Komponenten zwei Längen bzw. eine Länge und ein Winkel sind. Um die durch eine Variable tatsächlich angenommene Variante zu identifizieren, wird eine dritte Komponente, das sogenannte *Diskriminatorfeld*, eingeführt.

Beispiel: Im oben vorgestellten Bibliothekskatalog ist ein Feld Ausleiher nur sinnvoll, wenn der Status auf ausgeliehen gesetzt ist.

```

BuchT = RECORD
    Autor          : AutorT;
    Titel          : TitelT;
    Signatur       : SignT;
    ZahlBestelle: BestZahlT;
(* Programm P2.41a *)

```



```

Besteller      :      BestListenT;
Status         :      StatusT;
Ausleiher      :      LeserIDT;
Interessent    :      LeserIDT;
CASE Status    : StatusT OF
  | ausgeliehen : Ausleiher   : LeserIDT
  | bestellt    : Interessent : LeserIDT
END (* CASE *);
END (* RECORD *);

```

Bei der Bildung von Varianten-Records sind folgende *Regeln* zu beachten:

- alle Selektoren im Record müssen verschieden sein
- der Diskriminator kann in Sonderfällen auch wegfallen
- wie in der CASE-Anweisung kann auch hier eine leere Variante stehen (senkrechter Strich vor der ersten echten Variante).

Records mit Varianten werden verwendet, um die logische Klarheit zu erhöhen und um Speicher zu sparen. Beides soll nachfolgend erläutert werden.

Erhöhung der Klarheit

Ein Programm wird einem Leser verständlicher sein, wenn er den *logischen Zusammenhang* der Felder bereits aus der Typdefinition erkennen kann.

Beispiel: Beim Bibliothekskatalog hätte z.B. eine weitere Variante bezüglich einer evtl. Vorbestellung gebildet werden können. Das Feld vorbestellt muß dann natürlich auch in den Anweisungen berücksichtigt werden.

```

BuchT          = RECORD                                     (* Programm P2.41b *)
  ...
  CASE vorbestellt : BOOLEAN OF
    | FALSE : (* leer *)
    | TRUE  : ZahlBesteller : BestZahlT;
              Besteller     : BestListenT;
  END (* CASE *);
  ...
END (* RECORD *);

```

Speichereinsparung

Durch Verwendung von Varianten kann man den Speicherbedarf reduzieren, denn pro Record wird nur soviel Platz belegt, wie sich aus der Kombination der *invarianten* und der jeweils größten *varianten* Teile ergibt. Der Effekt ist nur signifikant, wenn es mehrere Varianten mit hohem Speicherbedarf gibt.

```

(* Programm P2.42 *)
TYPE ExternDatenBlatt = ARRAY [1..100] OF REAL;
InternDatenblatt = ARRAY [1..200] OF INTEGER;
Datenblatt
    = RECORD
        (* Varianten zur Speicherplatzeinsparung.*)
        Teilenummer: CARDINAL;
        CASE extern : BOOLEAN OF
            | FALSE :
                BlattInt : InternDatenblatt
            | TRUE :
                BlattExt : ExternDatenBlatt
        END (* CASE *);
    END (* RECORD *);

```

In vielen Fällen bringen Varianten gleichzeitig beide Vorteile, Klarheit und Speichereinsparung, wie folgendes Programm-Fragment zeigt.

```

TYPE KoordTyp = RECORD
    (* Programm P2.43 *)
    (* Varianten zur Strukturierung u. Speichereinsparung *)
    CASE polar : BOOLEAN OF
        | FALSE : x, y : REAL (* Kartesische Koordinaten *)
        | TRUE : r, phi : REAL (* Polar-Koordinaten *)
    END (* CASE *);
END (* RECORD *);
VAR Koord : ARRAY [1..1000] OF KoordTyp;
CONST Pi = 3.1415;

BEGIN
    WITH Koord [1] DO polar := FALSE; x := 1.0; y := -1.0;
    END (* WITH *);
    WITH Koord [2] DO polar := TRUE; r := 2.0; phi := Pi / 4.0;
    END (* WITH *);
END;

```

Gültigkeitsbereiche in Records

Es gelten folgende Regeln bzgl. der *Gültigkeitsbereiche* in Records:

- der Gültigkeitsbereich von neu definierten *Selektoren* im Record beginnt mit der Definition im Record und endet im umgebenden Block
- eine WITH-Anweisung öffnet einen neuen Block, in dem die Selektoren des entsprechenden Records (analog den lokalen Variablen einer Prozedur) gültig sind.

Beispiel:

```

MODULE RecordGB;
(* Programm P2.44 *)
(* Demonstration der Gültigkeitsbereiche in Varianten *)
FROM InOut IMPORT (* PROC *) Write, WriteLn;

```

```

VAR R1 : RECORD
    S1: RECORD
        S1: REAL;
        S3: BOOLEAN;
        END (* RECORD R1.S1 *);
    S2: CHAR;
    S3: INTEGER;
    END (* RECORD R1 *);
R2 : RECORD
    CASE T: BOOLEAN OF
        | TRUE : S1: CHAR;
                  S2: INTEGER;
                  S3: RECORD
                      S1: CHAR;
                      S3: CARDINAL;
                      END (* RECORD R2.S3 *);
        | FALSE: S4: INTEGER;
    END (* CASE *);
    END (* RECORD R2 *);
S1 : CHAR;
S2 : CARDINAL;

BEGIN (* RecordGB *)
    WITH R1 DO
        WITH R2.S3 DO
            S1 := 'a'; S2:= 'x'; S3 := MAX(CARDINAL);
        END (* WITH R2.S3 *);
    END (* WITH R1 *);
    WITH R1.S1 DO S1 := FLOAT (S2); S3 := R1.S2 < 'Z' END;
    S1 := '$!'; R2.S1 := '&';
    R2.T := FALSE;
    WITH R2 DO S1 := 'Q' END;
    Write(R2.S1); Write(S1); WriteLn;
END RecordGB.

```

Das vorstehende Programm wird ohne Fehlermeldung übersetzt, obwohl auf den ersten Blick nicht klar ist, daß die Typen überall *zuweisungskompatibel* sind. Um dies zu klären und das Ergebnis vorauszusagen, werden wir die WITH-Anweisungen aus und beseitigen sie:

```

BEGIN (* RecordGB *)
    R2.S3.S1 := 'a'; R1.S2:= 'x'; R2.S3.S3 := MAX(CARDINAL);
    R1.S1.S1 := FLOAT (S2); R1.S1.S3 := R1.S2 < 'Z';
    S1 := '$!'; R2.S1 := '&';
    R2.T := FALSE;
    R2.S1 := 'Q';
    Write(R2.S1); Write(S1); WriteLn;
END RecordGB.

```


Offenbar wird also Q\$ ausgegeben. Die Tatsache, daß bei der Ausführung R2.T mit FALSE besetzt und damit S1 eigentlich nicht zugänglich sein sollte, hat keinerlei Wirkung.

2.4.4 Zeiger und dynamische Variablen

Alle bisher verwendeten Variablen hatten einen Bezeichner und waren im sogenannten *Kellerspeicher* abgelegt. Dank der Felder und Verbunde sind wir damit auch in der Lage, komplexe, aber lediglich *statische* Datenstrukturen zu bilden. Kompliziertere Strukturen wie die in diesem Abschnitt vorgestellten Listen oder die in Abschnitt 6.2 behandelten Graphen (Netzwerke) und Bäume lassen sich so aber nicht beschreiben. Schon gar nicht ist es möglich, Datenstrukturen im Kellerspeicher *dynamisch*, also während der Programmausführung, zu verändern.

Dazu werden nun die sogenannten *dynamischen Variablen* eingeführt. Ihre Lebensdauer ist nicht wie die anderer Variablen an die Ausführung einer Prozedur gebunden, sondern sie werden zur Laufzeit explizit *erzeugt/kreiert* (mit der vordefinierten Prozedur ALLOCATE oder NEW) und eventuell auch wieder *beseitigt* (mit DEALLOCATE oder DISPOSE); sonst existieren sie bis zum Ende des Programmlaufs.

An dieser Stelle sei nochmals auf die Unterscheidung zwischen dem Gültigkeitsbereich des Bezeichners einer Variablen und der Lebensdauer des zur Laufzeit durch eine Variable belegten Speicherplatzes verwiesen (siehe Abschnitt 2.3.2).

Das Adjektiv "dynamisch" bedarf einer Erklärung: Auch eine Prozedur-lokale Variable ist dynamisch in dem Sinne, daß ihr Speicherplatz nur für die Dauer der Prozedur-Ausführung reserviert wird. Bei den "dynamischen" Variablen ist aber etwas anders gemeint: Die Dynamik liegt vollständig unter der Kontrolle des Programmierers, und ein statischer Aspekt fehlt ganz, weil diese Variablen keinen Bezeichner, also auch keinen Gültigkeitsbereich haben (siehe unten).

2.4.4.1 Die Speicherung auf der Halde

Wegen der freien Bestimmung der Lebensdauer können dynamische Variablen nicht im (mit den Prozedur-Inkarnationen pulsierenden) Kellerspeicher gespeichert werden. Für sie wird darum ein ganz anderer Speicher, die *Halde* (der *Heap*) zur Verfügung gestellt.

Da die dynamischen Variablen während des Programmablaufs in prinzipiell beliebiger Menge geschaffen werden können, ist es nicht möglich, ihnen (wie den Variablen im Kellerspeicher) *feste* Bezeichner zuzuordnen. Sie werden stattdessen durch *Zeiger*

(*Pointer*) identifiziert. Zeiger können sowohl im Kellerspeicher als auch auf der Halde liegen; die Variablen, auf die sie zeigen, liegen in jedem Falle auf der Halde.

Zeiger sind - wie alle anderen Variablen - zu Beginn ihrer Lebensdauer undefiniert. Sie können auf drei Arten einen *definierten* Wert erhalten:

- durch *Speicherbelegung* mit NEW oder ALLOCATE (siehe unten);
damit weist nur dieser Zeiger auf den frisch belegten Platz
- durch Wertzuweisung eines anderen Zeigers;
damit weisen beide Zeiger auf denselben Platz der Halde. Man beachte, daß die so bezeichnete dynamische Variable bei der Wertzuweisung nicht kopiert wird!
- durch Zuweisung von *NIL*,
eine für alle Zeigertypen vordefinierte Konstante, die *unbesetzte Zeiger* kennzeichnet.

Für das Verständnis und den Gebrauch dynamischer Variablen ist es notwendig, etwas über die Implementierung der Halde zu wissen. Die *Verwaltung* der Halde ist nicht wie die Kellerspeicherverwaltung in die Sprache MODULA-2 (d.h. in den Übersetzer und evtl. in das Laufzeitsystem) integriert, sondern muß vom Programmierer explizit durch Import der Prozeduren zur *Speicherbelegung* und -*freigabe* (normalerweise ALLOCATE und DEALLOCATE aus dem Modul Storage) angefordert werden. Dies gilt auch dann, wenn diese Prozeduren nur implizit (nämlich durch die in einigen MODULA-2-Systemen vordefinierten Prozeduren NEW und DISPOSE) aufgerufen werden.

Sei XT ein beliebiger Typ (oft ein Record), und seien PT und P definiert durch

```
TYPE PT = POINTER TO XT; VAR P : PT;
```

Dann sind die beiden Operationen

```
NEW (P);    DISPOSE (P);
```

gleichbedeutend mit

```
ALLOCATE (P, SIZE(XT));    DEALLOCATE (P, SIZE(XT));
```

SIZE ist dabei eine vordefinierte Funktion, die nicht importiert werden muß. NEW oder ALLOCATE belegt also auf der Halde Speicherplatz für ein Objekt des Typs XT (nicht PT!), DISPOSE oder DEALLOCATE gibt diesen Speicher wieder frei. Außerdem weist NEW dem Zeiger P als Wert die (Anfangs-) Adresse des belegten Speicherplatzes zu. Der Programmierer kann (und sollte) von dieser aber keinen Gebrauch anders als durch *Zeiger-Operationen* (folgt in Abschnitt 2.2.4.2) machen, denn sie kann sich je nach Implementierung von einem Lauf des Programms zum anderen verändern. Verfügt das System über ein Programm, das die Halde

“aufräumt” (einen sog. *Garbage Collector*), so ist sogar eine Änderung während des Laufs möglich, wobei der Garbage Collector automatisch alle Zeiger korrigiert. Der Zugriff auf den belegten Speicherplatz erfolgt mit dem sogenannten *Dereferenzieroperator* $\wedge$ (siehe folgende Beispiele).

Durch Ausführung von DISPOSE oder DEALLOCATE wird der Wert des betreffenden Zeigers wieder undefiniert. Bei manchen Systemen werden Zeiger zu Beginn und nach DISPOSE *automatisch* mit NIL besetzt. Da dies aber nicht auf allen Systemen gilt, leidet die Portabilität, wenn dieser Komfort in Anspruch genommen wird.

Auf einfachen MODULA-2-Systemen ist DISPOSE/DEALLOCATE meist wirkungslos; andererseits kann ein Garbage Collector evtl. sogar selbst erkennen, welcher Speicher freigegeben ist (weil keine Zeiger mehr darauf verweisen). Darum neigen Programmierer dazu, die Speicherfreigabe zu “vergessen”. Dies ist schlechter Stil, weil es den Dokumentationswert des Programms vermindert und dem Garbage Collector komplizierte Untersuchungen aufzwingt, die mit wenig Überlegung beim Programmieren überflüssig werden. Es ist etwa das gleiche, als ob man in einer Bibliothek die gelesenen Bücher einfach irgendwo herumliegen läßt, statt (durch Ablegen auf dem Rückgabetisch) dem Personal explizit zu zeigen, daß diese wieder allgemein verfügbar sind.

Das Verhalten des Systems bei Überlauf der Halde ist nicht definiert. Der Programmierer kann dem aber vorbeugen, indem er vor der Belegung mit der (ebenfalls aus dem Modul Storage zu importierenden) Funktion *Available* abfragt, ob noch genügend Speicher verfügbar ist.

Bemerkung: Das Thema “Speicherverwaltung” insgesamt werden wir noch ausführlich in Abschnitt 6.6 behandeln.

Die Beziehung zwischen Kellerspeicher und Halde läßt sich durch einen Vergleich veranschaulichen (siehe auch Abb. 2.17): Der Kellerspeicher entspricht dem festen Boden, und die Variablen sind schwere, auf dem Boden fixierte Gegenstände. Die Halde ist die Luft, und die dynamischen Variablen sind Ballons. Ballons müssen direkt oder über andere Ballons mit dem Boden (von dort durch den Anker mit dem untersten Ballon) verbunden sein, sonst steigen sie auf oder werden abgetrieben. Die Seile, durch die sie verbunden sind, entsprechen den Zeigern. In der Realität des Ballons bleibt seine genaue Position unbestimmt, sie schwankt mit dem Wind. Ebenso ist die Adresse einer dynamischen Variablen unbestimmt, und nur ein im Kellerspeicher ruhender Zeiger (oder eine im Kellerspeicher beginnende Kette von Zeigern) macht sie erreichbar.

Abb. 2.17 Vergleich dynamischer Variablen mit Ballons

2.2.4.2 Operationen auf Zeigern

Das nachfolgende Programm P2.45 zeigt einige einfache Zeiger-Operationen.

```

MODULE Pointer;                                     (* Programm P2.45 *)
(* Beispiel zu den Zeigervariablen *)
FROM InOut  IMPORT (* PROC *) WriteCard, WriteString, WriteLn;
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
TYPE ZeigTyp = POINTER TO ZeigRec;
   ZeigRec = RECORD
       Key : CARDINAL;
       Next: ZeigTyp;
       (* Cont: ... *)
   END (* RECORD *);
VAR Anker, Hilfl : ZeigTyp;
BEGIN (* Pointer *)                                (* Abb.bei BEGIN *)
  (* Kette aus zwei Gliedern aufbauen *)
  NEW (Anker);                                     (* Abb.nach erstem NEW *)
  WITH Anker^ DO
    Key := 1; NEW (Next);                          (* Abb.nach zweitem .NEW *)
    WITH Next^ DO Key := 2; Next := NIL; END;      (* Abb.nach Next := NIL *)
  END (* WITH Anker^ *);
  (* Kette durchlaufen und Werte von Key ausgeben *)
  Hilfl := Anker;                                  (* Hilfszeiger initialisieren *)
  WHILE Hilfl # NIL DO                             (* Test Kettenende *)
    WriteCard (Hilfl^.Key, 3);                     (* Ausgabe des Schluessels *)
    Hilfl := Hilfl^.Next;                          (* Weitersetzen des Hilfszeigers *)
  END (* WHILE *);
  WriteLn;                                          (* Die erzeugte Ausgabe ist: 1 2 *)
END Pointer.

```

Der linke Teil der Abb. 2.18 zeigt die Situation zu Beginn des Programmlaufs; es existiert nur ein undefinierter Zeiger Anker. Der Zeiger Hilf1, der ebenfalls undefiniert ist, spielt in diesem Teil des Programms keine Rolle, er wird daher weggelassen. Durch NEW erhält der Anker einen definierten Wert, er weist nun in die Halde auf einen Record, dessen Felder ihrerseits undefiniert sind (rechter Teil der Abb. 2.18).

Abb. 2.18 Situation bei BEGIN und nach erstem NEW

Die Wertzuweisung an Key gibt der dynamischen Variablen einen Schlüssel (1), durch NEW wird ein weiterer Record angehängt (linker Teil der Abb. 2.19). Die Felder dieses Records werden anschließend mit dem Schlüssel 2 und NIL besetzt (rechter Teil der Abb. 2.19).

Abb. 2.19 Situation nach zweitem NEW und nach Next := NIL

Das restliche Programm läßt den Hilfszeiger Hilf1 von einer dynamischen Variablen zur folgenden laufen, bis das durch NIL definierte Ende erreicht ist. Bei jedem Schritt wird der Inhalt des Feldes Key ausgegeben. Hilf1 durchläuft also die Werte:

zu Beginn	undefiniert
nach Initialisierung mit Wert von Anker	Adresse von Record 1
nach dem 1. Schleifendurchlauf	Adresse von Record 2
nach dem 2. Schleifendurchlauf	NIL.

Dann ist die Wiederholbedingung nicht mehr erfüllt.

Nachfolgend soll die Wirkung von vier verschiedenen Anweisungen ((a) - (d)) gezeigt werden, die jeweils (alternativ, nicht sequentiell) von der durch das Beispiel oben geschaffenen Situation (rechter Teil der Abb. 2.19) ausgehen.

`Anker^.Next^.Next := Anker;` (* a *)

Das bisher mit NIL besetzte Ende der Kette wird auf das 1. Element gerichtet (linker Teil der Abb. 2.20). Diese sogenannte *zyklische Verkettung* ist in vielen Anwendungen verketteter Listen vorteilhaft. Natürlich lautet das Ende-Kriterium nun nicht mehr `Hilf1 = NIL`, sondern `Hilf1 = Anker`.

`Anker := Anker^.Next^.Next;` (* b *)

Die rechte Seite der Wertzuweisung hat den Wert NIL, und dieser wird dem Anker zugewiesen. Damit werden die beiden dynamischen Variablen unerreichbar, sie sind *implizit freigegeben* (rechter Teil der Abb. 2.20).

Eine *explizite* Freigabe könnte beispielsweise erfolgen durch:

`DISPOSE (Anker^.Next); DISPOSE (Anker); Anker := NIL;`
`Anker := Anker^.Next;` (* c *)

Abb. 2.20 Wirkung der Anweisung (a)

Wirkung der Anweisung (b)

Der Anker weist durch Anweisung c auf den zweiten Record. Der erste ist damit implizit freigegeben (linker Teil in Abb. 2.21); er enthält zwar selbst noch einen Zeiger auf den 2. Record, doch macht ihn dieser Zeiger umgekehrt nicht zugänglich.

NEW (Anker);

(* d *)

Auch hier sind die beiden Records implizit freigegeben, und ein neuer mit undefiniertem Inhalt ist durch Anker bezeichnet (rechter Teil in Abb. 2.21).

Abb. 2.21 Wirkung der Anweisung c

Wirkung der Anweisung d

2.4.4.3 Verkettete Listen

Einfach verkettete Listen

Im Beispiel des letzten Abschnitts war eine kurze Liste entstanden. In der Praxis haben Listen i.a. natürlich eine *undefinierte* Länge, denn der Vorteil der dynamischen Variablen liegt ja gerade in der Freiheit, Strukturen dynamisch zu verändern.

Nachfolgend werden der Aufbau einer einfach verketteten Liste sowie das Löschen und Einfügen von Listenelementen vorgestellt.

Aufbau einer Liste

Beim Aufbau einer allgemeinen, einfach verketteten Liste (im folgenden kurz *Kette* genannt) werden die Elemente der Kette vom Anker an nummeriert. Die Typen sind gleich geblieben, die Ausgabeprozedur *ZeigKette* funktioniert wie im letzten

Abschnitt, doch wird zusätzlich eine Meldung ausgegeben, die auf feste Länge aufgefüllt wird.

```

MODULE Kette;                                     (* Programm P2.46a *)
(* Beispiel 2 zu Zeigervariablen: Aufbau einer verketteten
   Liste; hier werden nicht NEW/ DISPOSE, sondern ALLOCATE/
   DEALLOCATE verwendet *)
FROM InOut   IMPORT (* PROC *) WriteCard, WriteString,
                    WriteChar, WriteLn;
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
TYPE ZeigTyp = POINTER TO ZeigRec;
   ZeigRec = RECORD
       Key : CARDINAL;
       Next: ZeigTyp;
       (* Cont: ... *)
   END (* RECORD *);
VAR Anker : ZeigTyp;

PROCEDURE ZeigKette (Meldung: ARRAY OF CHAR);
(* Meldung in konstanter Laenge ausgeben,
   dann Kette (ist global) *)
CONST LngMeldMax = 30;
VAR Hilfl : ZeigTyp;   i: CARDINAL;
BEGIN (* ZeigKette *)
    WriteString (Meldung);
    FOR i := HIGH (Meldung)+1 TO LngMeldMax-1 DO
        WriteChar(' ');
    END (* FOR *);
    Hilfl := Anker;
    WHILE Hilfl # NIL DO
        WriteCard (Hilfl^.Key, 3);   Hilfl := Hilfl^.Next;
    END (* WHILE *);
    WriteLn;
END ZeigKette;

PROCEDURE BaueKettel (Lng: CARDINAL);
(* Kette unbest. Laenge > 0 aufbauen
   durch Anfuegen am Ende *)
VAR Hilfl : ZeigTyp;   i : CARDINAL;
BEGIN (* BaueKettel *)
    ALLOCATE (Anker, SIZE(ZeigRec));
    Hilfl := Anker;   Anker^.Key := 1;   (* 1. Kettenelement *)
    FOR i := 2 TO Lng DO           (* Kettenelemente 2 bis Lng *)
        ALLOCATE (Hilfl^.Next, SIZE (ZeigRec));
        Hilfl := Hilfl^.Next;   Hilfl^.Key := i;
    END (* FOR *);
    Hilfl^.Next := NIL;           (* Abschluss der Kette mit NIL *)
END BaueKettel;

BEGIN (* Kette *)
    BaueKettel (3);               ZeigKette ('Aufbau der Laenge 3');
END Kette.

```

Abb. 2.22 Aufbau der Kette mit BaueKette1

Abb. 2.22 zeigt die Situation nach Ausführung von BaueKette1 (3). Da wir uns mit Hilf1 an der Kette "entlanghangeln", können wir auch ohne die Laufvariable i auskommen. Diese Lösung (Programm P2.46b) ist aber deutlich komplizierter und darum schlechter als die oben gezeigte.

```

PROCEDURE BaueKette2 (Lng: CARDINAL);      (* Programm P2.46b *)
(* Gleiche Funktion wie BaueKettel, aber ohne Laufvariable *)
VAR Hilf1 : ZeigTyp;
BEGIN (* BaueKette2 *)
  ALLOCATE (Anker, SIZE (ZeigRec));
  Hilf1 := Anker;  Anker^.Key := 1;
  WHILE Hilf1^.Key < Lng DO
    ALLOCATE (Hilf1^.Next, SIZE (ZeigRec));
    Hilf1^.Next^.Key := Hilf1^.Key + 1;  Hilf1 := Hilf1^.Next;
  END (* WHILE *);
  Hilf1^.Next := NIL;
END BaueKette2;

```

Statt mit dem ersten kann man auch mit dem letzten Element beginnen. Das ergibt eine einfachere Lösung, die auch den Fall der leeren Liste ohne spezielle Behandlung einschließt. Der Trick besteht darin, daß am Hilfszeiger jeweils nur ein einzelnes Element erzeugt und dann zwischen Anker und Kette eingehängt wird (siehe Programm P2.46c).

```

PROCEDURE BaueKette3 (Lng: CARDINAL);      (* Programm P2.46c *)
(* Aufbau der Kette von hinten nach vorn *)
VAR Hilf1 : ZeigTyp;  i : CARDINAL;
BEGIN (* BaueKette3 *)
  Anker := NIL; (* Kette initialisieren *)
  FOR i := Lng TO 1 BY -1 DO      (* rueckwaerts wegen Key *)
    ALLOCATE (Hilf1, SIZE (ZeigRec));  (* Neues Element
                                         erzeugen *)
    WITH Hilf1^ DO Key := i; Next := Anker; END; (* und
                                         besetzen *)
    Anker := Hilf1;  (* Neues Element wird vor die Kette
                                         gehaengt *)
  END (* FOR *);
END BaueKette3;

```


Abb. 2.23 zeigt die Situation beim Einfügen des drittletzten Elementes (mit Schlüssel 5) in die Kette (gestrichelt der neue Wert von Anker).

Abb. 2.23 Aufbau der Kette rückwärts mit BaueKette3

Verkettete Listen sind offensichtlich *rekursive* Datenstrukturen, denn jedes Element der Liste enthält einen Zeiger auf eine Liste. Darum lassen sie sich auch besonders gut mit rekursiven Prozeduren bearbeiten. Bei der Arbeit mit sehr langen Ketten stehen dem in der Praxis allerdings Effizienzüberlegungen entgegen. Auch kann es zum *Überlauf* des Kellerspeichers kommen, denn an jedes Listenelement wird ja eine Prozedur-Inkarnation geheftet. Wo aber rekursive Lösungen praktisch gebraucht werden können, sind sie wesentlich eleganter als die iterativen.

Man beachte, daß der Zeiger im folgenden Beispiel als Referenz-Parameter übergeben wird. Fast nie hat es Sinn, Zeiger als Wert-Parameter zu übergeben, denn dies führt dazu, daß eine Kopie des Zeigers angelegt wird. Wir wollen aber das Original des Zeigers verändern (beispielsweise durch NEW). Den Vorteil des Wert-Parameters, nämlich die *Entkopplung* zwischen Original und Kopie, haben wir bei Zeigern ohnehin nicht, denn es wird ja nur der Zeiger, nicht aber die eigentlich wichtige Information in der dynamischen Variablen kopiert. Man beachte, daß uns die Parameter-Übergabe die Möglichkeit verschafft, dynamische Variablen wieder über einen Bezeichner anzusprechen, nämlich den des *formalen Parameters*.

```
(* Programm P2.46d *)
PROCEDURE BaueKette4 (    Laenge, Schluessel: CARDINAL;
                        VAR Zeiger           : ZeigTyp);
(* Aufbau einer Kette durch Rekursion; keine lokale Variable *)
BEGIN (* BaueKette4 *)
  IF Laenge = 0 THEN
    Zeiger := NIL;
  ELSE
    ALLOCATE (Zeiger, SIZE (ZeigRec));
    WITH Zeiger^ DO
      Key := Schluessel;
      BaueKette4 (Laenge-1, Schluessel+1, Next);
    END (* WITH *);
  END (* IF *);
END BaueKette4;
```

Abb. 2.24 zeigt den Aufbau der Kette mit BaueKette4. Die Zeiger, die als Parameter übergeben werden, aber bis auf den ersten keinen Bezeichner haben, sind durch griechische Buchstaben identifiziert. Der formale Parameter Zeiger ist also in der ersten Inkarnation von BaueKette4 mit α = Anker besetzt, in der zweiten mit β = Anker[^].Next usw. In der gezeichneten Situation beim Abbruch der Rekursion wird gerade der letzte Zeiger (ω) mit NIL besetzt.

Abb. 2.24 Rekursiver Aufbau einer Kette mit BaueKette4

Löschen in einer Liste

Nachfolgend stellen wir Algorithmen zum *Löschen* von Elementen in einer Kette vor. Um das zu löschende Element explizit freigeben zu können, benötigen wir einen weiteren Hilfszeiger. Damit weist Hilf1 auf das "Opfer", Hilf2 auf den Vorgänger. Das Programm P2.46e ist den unten beschriebenen Sonderfällen (Löschen des ersten Elementes oder zu löschendes Element nicht in der Liste) nicht gewachsen.

```

PROCEDURE Loeschen1 (Nr: CARDINAL);          (* Programm P2.46e *)
VAR Hilf1, Hilf2 : ZeigTyp;
BEGIN (* Loeschen1 *)
  Hilf1 := Anker;
  REPEAT                                     (* Suche nach dem "Opfer" *)
    Hilf2 := Hilf1;                         (* Hilf1 weist auf den Kandidaten, *)
    Hilf1 := Hilf1^.Next; (* ... Hilf2 auf dessen Vorgaenger *)
  UNTIL

```



```

UNTIL Hilfl1^.Key = Nr;
Hilf2^.Next := Hilfl1^.Next;  (* damit faellt Hilfl1^ aus
                                der Kette *)
DEALLOCATE (Hilfl1, SIZE (ZeigRec));
(* Hilfl1^ wird ordnungsgemaess "kompostiert" *)
END Loeschen1;

```

Mit Loeschen1 gibt es keine Schwierigkeiten, wenn das Opfer nicht am Anfang steht. Dagegen lässt sich das erste (oder einzige) Element der Kette so nicht löschen. Wir müssen also diesen Fall separat behandeln (siehe Programm P2.46f).

```

PROCEDURE Loeschen2 (Nr: CARDINAL);          (* Programm P2.46f *)
(* Wie Loeschen1, jedoch auch fuer 1. Listenelement korrekt *)
VAR Hilfl1, Hilfl2 : ZeigTyp;
BEGIN (* Loeschen2 *)
  Hilfl1 := Anker;
  IF Anker^.Key = Nr THEN
    Anker := Anker^.Next;  (* Sonderfall: 1. Element wird
                            geloescht *)
  ELSE
    (* wie in Loeschen1 *)
    REPEAT
      Hilfl2 := Hilfl1; Hilfl1 := Hilfl1^.Next;
    UNTIL Hilfl1^.Key = Nr;
    Hilfl2^.Next := Hilfl1^.Next;
  END (* IF *);
  DEALLOCATE (Hilfl1, SIZE (ZeigRec));  (* in jedem Falle
                                          noetig *)
END Loeschen2;

```

Doch auch dieses Programm ist nicht korrekt: Ist das zu löschende Element nicht in der Liste vorhanden, so bricht das Programm mit einem Laufzeitfehler ab, da versucht wird, auf das Element eines Zeigers zuzugreifen (*Dereferenzieren*), dieser aber den Wert NIL hat. Deshalb erfordert auch dieser Fall eine spezielle Behandlung, die in Programm P2.46g realisiert ist.

```

PROCEDURE Loeschen3 (Nr: CARDINAL);          (* Programm P2.46g *)
(* Wie Loeschen2, jedoch auch korrekt, falls angegebenes
  Element fehlt *)
VAR Hilfl1, Hilfl2 : ZeigTyp;
BEGIN (* Loeschen3 *)
  IF Anker # NIL THEN
    (* bei leerer Liste keine Aktion *)
    Hilfl1 := Anker;
    (* wie Loeschen2 *)
    IF Anker^.Key = Nr THEN
      Anker := Anker^.Next; DEALLOCATE (Hilfl1, SIZE(ZeigRec));
    ELSE
      REPEAT
        Hilfl2 := Hilfl1; Hilfl1 := Hilfl1^.Next
      UNTIL (Hilfl1 = NIL) OR (Hilfl1^.Key = Nr);
      (* hier ist auch die erfolglose Suche abgedeckt *)
      IF Hilfl1 # NIL THEN
        Hilfl2^.Next := Hilfl1^.Next;

```



```

        DEALLOCATE (Hilf1, SIZE (ZeigRec));
    END (* IF *);
END (* IF *);
END (* IF *);
END Loeschen3;

```

Die Reihenfolge der logischen Operanden des OR-Operators in der UNITL-Bedingung ist wichtig: Ergibt die Auswertung des ersten Operanden bereits TRUE, so wird der zweite Operand nicht mehr ausgewertet, da der Wert des gesamten Ausdrucks dann bereits feststeht. Durch diese *Auswertungsreihenfolge* logischer Operanden wird der oben beschriebene Laufzeitfehler durch das Dereferenzieren eines NIL-Zeigers vermieden. Analoges gilt auch für den AND-Operator: Die Auswertung der Operanden erfolgt von links nach rechts und wird abgebrochen, falls die Auswertung eines Operanden FALSE ergibt.

Eine andere, oft vorteilhafte Lösung ist die, ein "nulltes" Element hinzuzufügen, das nie verändert wird, sondern nur eine homogene Adressierung aller anderen Elemente der Kette ermöglicht; es bildet damit so etwas wie einen *fliegenden Anker* (angewandt im Programm P2.48).

Auch beim Löschen hat die Rekursion Vorteile, wie die folgende Prozedur zeigt. Der Sonderfall "erstes Element löschen" ist ohne spezielle Behandlung abgedeckt, "leere Liste" und "Element nicht vorhanden" erfordern zusammen nur eine Abfrage am Anfang.

```

(* Programm P2.46h *)
PROCEDURE Loeschen4 (Nr:CARDINAL; VAR Zeiger: ZeigTyp);
(* Loeschen als rekursive Prozedur *)
VAR Hilf1 : ZeigTyp;          (* nur e i n Hilfszeiger *)
BEGIN (* Loeschen4 *)
    IF Zeiger # NIL THEN      (* leere Liste oder nicht gefunden
                                abfangen *)
        IF Zeiger^.Key = Nr THEN
            Hilf1 := Zeiger;  Zeiger := Zeiger^.Next;
            DEALLOCATE (Hilf1, SIZE (ZeigRec));
        ELSE
            Loeschen4 (Nr, Zeiger^.Next); (* in restl. Liste suchen *)
        END (* IF *);
    END (* IF *);
END Loeschen4;

```

Hier folgt das vollständige Hauptprogramm im Modul Kette, durch das alle Prozeduren zum Aufbau und zum Löschen getestet werden :

```

(* Programm P2.46i *)
BEGIN (* Kette *) (* Hauptprogramm fuer Listen-Algorithmen *)
    BaueKettel (5);    ZeigKette ('Aufbau vorwaerts 1');
    BaueKette2 (6);    ZeigKette ('Aufbau vorwaerts 2');

```

```

BaueKette3 (7);      ZeigKette ('Aufbau rueckwaerts');
BaueKette4 (8, 1, Anker);
                        ZeigKette ('Aufbau rekursiv');
Loeschen1 (5);      ZeigKette ('Element in Liste loeschen');
Loeschen1 (8);      ZeigKette ('Element am Ende loeschen');
Loeschen2 (1);      ZeigKette ('Element am Anfang loeschen');
Loeschen3 (9);      ZeigKette ('Versuch, fehl. El. loeschen');
Loeschen4 (2, Anker);
                        ZeigKette ('rekursiv am Anfang loeschen');
Loeschen4 (7, Anker);
                        ZeigKette ('rekursiv am Ende loeschen');
END Kette.

```

Die Ausgabe des Programms sieht so aus:

Aufbau vorwaerts 1	1	2	3	4	5		
Aufbau vorwaerts 2	1	2	3	4	5	6	
Aufbau rueckwaerts	1	2	3	4	5	6	7
Aufbau rekursiv	1	2	3	4	5	6	7 8
Element in Liste loeschen	1	2	3	4	6	7	8
Element am Ende loeschen	1	2	3	4	6	7	
Element am Anfang loeschen	2	3	4	6	7		
Versuch, fehl. El. loeschen	2	3	4	6	7		
rekursiv am Anfang loeschen	3	4	6	7			
rekursiv am Ende loeschen	3	4	6				

Einfügen in eine Liste

Das *Einfügen* läßt sich ganz ähnlich wie das Löschen realisieren. Wir brauchen zwei Hilfszeiger, von denen der eine auf den Vorgänger, der andere auf den Nachfolger des zu einzufügenden Elementes gerichtet wird. Dann kann am Vorgänger das neue Element erzeugt und der Rest angehängt werden. Auch hier ist es vorteilhaft, wenn entweder nie vor dem bisher ersten Element eingefügt oder eine rekursive Realisierung gewählt wird. Die hier gezeigte Lösung enthält keine Prüfung darauf, ob der Schlüssel bereits vergeben wurde. Dafür wären ein modifizierter Test $\text{Key} \geq \text{Nr}$ und eine weitere Abfrage im THEN-Zweig notwendig.

```

(* Programm P2.47 *)
PROCEDURE Einfuegen (Nr: CARDINAL; VAR Zeiger: ZeigTyp);
(* Rekursive Loesung *)
VAR Hilf1 : ZeigTyp;
BEGIN (* Einfuegen *)
  IF (Zeiger = NIL) OR (Zeiger^.Key > Nr)
  THEN (* Stelle zum Einfuegen gefunden *)
    NEW (Hilf1);
    WITH Hilf1^ DO Key := Nr; Next := Zeiger; END;
    Zeiger := Hilf1;
  ELSE
    Einfuegen (Nr, Zeiger^.Next);
  END (* IF *);
END Einfuegen;

```


Der Aufruf erfolgt beispielsweise mit

```
Einfuegen (7, Anker);
```

Doppelt verkettete Listen

In vielen Fällen ist es vorteilhaft, Listen *doppelt* zu verketten. Zwar steigt der *Speicherbedarf* durch die zusätzlichen Zeiger auf den *Vorgänger* (Prev für Previous) an, doch werden dafür die Operationen einfacher, denn man kann sich in beiden Richtungen bewegen und auch eine Verbindung auftrennen, ohne den Zusammenhang zu verlieren. Um nicht Anfang und Ende der Kette separat behandeln zu müssen, verwendet man zweckmäßig den fliegenden Anker, also ein *Blindelement* der Kette.

Abb. 2.25 Doppelt verkettete Liste mit Blindelement

Abb. 2.25 zeigt eine solche Kette im Anfangszustand (also leere Kette, nur Blindelement vorhanden) und mit einem Element. Man beachte, daß es damit beim Einfügen und Löschen keine Sonderfälle (erstes oder letztes Element, Kette leer) mehr gibt.

Das folgende *Beispiel* verwendet direkt ALLOCATE und DEALLOCATE.

```
MODULE DoppelKette;                                     (* Programm P2.48 *)
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
FROM InOut  IMPORT (* PROC *) WriteCard, WriteLn;
TYPE ZeigTyp = POINTER TO ZeigRec;
   ZeigRec = RECORD
       Key : CARDINAL;
       Prev, Next: ZeigTyp;
       (* Cont: ... *)
   END (* RECORD *);
VAR Anker : ZeigTyp;
```



```

PROCEDURE Einfuegen (Nr: CARDINAL);
(* Fuegt neues Element in die geordnete Liste ein. Element mit
   Key = Nr darf noch nicht vorhanden sein *)
VAR Hilfl : ZeigTyp;
BEGIN (* Einfuegen *)
  Hilfl := Anker^.Next;
  WHILE (Hilfl^.Key < Nr) AND (Hilfl <> Anker) DO
    Hilfl := Hilfl^.Next; (* Nachfolger des neuen Elementes *)
  END (* WHILE *);          (* ... oder Kettenende suchen *)
  WITH Hilfl^ DO            (* der Nachfolger des neuen Elementes *)
    ALLOCATE (Prev^.Next, SIZE (ZeigRec)); (* neues Elem.
                                           einfuegen *)

    WITH Prev^.Next^ DO
      Key := Nr;
      Next := Hilfl;
      Prev := Hilfl^.Prev;
    END (* WITH *); (* damit ist das neue Element
                    ganz definiert *)

    Prev := Prev^.Next; (* neuen Vorgaenger anhaengen *)
  END (* WITH *);
END Einfuegen;

PROCEDURE Loeschen (Nr: CARDINAL);
(* loescht Element in Liste.
   Element mit Key = Nr muss vorhanden sein *)
VAR Hilfl : ZeigTyp;
BEGIN (* Loeschen *)
  Hilfl := Anker^.Next;
  WHILE Hilfl^.Key # Nr DO Hilfl := Hilfl^.Next; END;
                                           (* suchen *)

  WITH Hilfl^ DO
    Prev^.Next := Next; Next^.Prev := Prev;
  END (* WITH Hilfl^ *);
  DEALLOCATE (Hilfl, SIZE (ZeigRec)); (* loeschen *)
END Loeschen;

PROCEDURE ZeigKette;
VAR Hilfl : ZeigTyp;
BEGIN (* ZeigKette *)
  Hilfl := Anker^.Next;
  WHILE Hilfl # Anker DO
    WriteCard (Hilfl^.Key, 3); Hilfl := Hilfl^.Next;
  END (* WHILE *);
  WriteLn;
END ZeigKette;

BEGIN (* DoppelKette *)
  ALLOCATE (Anker, SIZE (ZeigRec)); (* fliegenden Anker init. *)
  WITH Anker^ DO Key := 0; Next := Anker; Prev := Anker; END;
  Einfuegen(7); Einfuegen(1); Einfuegen(4);
  Einfuegen(5); Einfuegen(9); ZeigKette;
  Loeschen (1); Loeschen (9); Loeschen (5); ZeigKette;
END DoppelKette.

```

2.4.4.4 Anwendungen und Probleme dynamischer Variablen

Dynamische Variablen werden vor allem aus drei Gründen verwendet:

- Aufbau *beliebiger* dynamisch *änderbarer* Datenstrukturen
- von Elementen mit hohem oder unbestimmtem Speicherbedarf aus ansonsten "regelmäßigen" Datenstrukturen wie Arrays und Records
- Realisierung sogenannter *Abstrakter Datentypen* (werden in Kapitel 3 vorgestellt).

Für den ersten Fall wurden Listen bereits als Beispiele gezeigt; weitere folgen in Kapitel 6 (z.B. Baumstrukturen). Den zweiten Fall soll das folgende Beispiel anschaulich machen: Es sei eine sehr große, schwach besetzte Matrix gegeben. Dort, wo Felder besetzt sind, sei der Speicherbedarf hoch. Wir könnten nun die Matrix wie folgt aufbauen:

```

TYPE MatrixElement = RECORD                                (* Programm P2.49 *)
    CASE vorhanden : BOOLEAN OF
        | FALSE : (* leer *)
        | TRUE  : Inhalt : GrosserRecordTyp
    END (* CASE *)
END (* RECORD *);
VAR Matrix : ARRAY [1..100], [1..100] OF MatrixElement;
```

Das führt aber dazu, daß *alle* Elemente für den ungünstigeren Fall (TRUE) ausgelegt werden müssen. Darum ist es oft vorteilhaft, in der Matrix nur Zeiger zu speichern:

```

TYPE MatrixElement = POINTER TO GrosserRecordTyp;
```

Die Matricelemente werden mit NIL initialisiert, und nur für diejenigen, die wirklich besetzt sind, wird (in der Halde) Speicherplatz belegt.

Probleme

Neben den genannten Vorteilen bringen dynamische Variablen auch *Nachteile* mit sich. Dazu zählen:

- Programme mit dynamischen Variablen lassen sich schwer *testen*.
Der Grund liegt darin, daß Bezeichner, die sich ausgeben ließen, fehlen. Man kann hier jedoch durch bewußte Testplanung (vgl. Abschnitt 4.3) vieles erleichtern.
- Programme mit dynamischen Variablen sind schwerer *verständlich* und *wartbar*.
Natürlich kann ein Programmierer dieses Problem mit den üblichen Mitteln (gute Bezeichner, Kommentare, Gliederung, Dokumentation) wesentlich entschärfen.

- Dynamische Variablen lassen sich nur mit beträchtlichem Aufwand auf *externe Datenspeicher* bringen oder von solchen einlesen.

Praxisnahe Programme arbeiten in fast allen Fällen mit Daten, die in externen Datenbanken oder in *Dateien* (Vorstellung im nächsten Abschnitt) gespeichert sind. Da auf solche Dateien mit Schreib- und Leseoperationen zugegriffen wird, wie wir sie ähnlich schon bisher verwendet haben, können wir Zeiger nicht auf diese Weise aus- und eingeben.

Das Problem läßt sich nicht elegant lösen. Man muß entweder die Datenstruktur mit speziellen Programmen so ausgeben, daß sie sich (dank Hilfsinformationen) später *rekonstruieren* läßt (bei einfachen Listen eine triviale Aufgabe), oder man implementiert sich die Halde selbst (und damit im Kellerspeicher), indem man alle eigentlich dynamischen Variablen in ein großes *Feld* legt. An die Stelle der Zeiger treten dann Indizes. Die Ein- und Ausgabe des Feldes ist dann völlig unproblematisch. Dafür muß man die Speicherverwaltung selbst implementieren.

In speziellen Fällen wird man das Problem durch hardware-nahe Lösungen angehen, die hier nicht behandelt werden können.

Das folgende *Beispiel* zeigt eine Variante des Programms zum Aufbau einer Kette, wobei die Halde simuliert wird. Die Prozedur BaueKette ist an BaueKette3 (Programm P2.46c) angelehnt. New und Dispose treten hier an die Stelle von NEW und DISPOSE. Die Verwaltung des Leerspeichers hat natürlich erst Sinn, wenn dieser nicht mehr wie hier im Anfangszustand ist.

```

MODULE KetteMitFeld;                                (* Programm P2.50 *)
(* Verkettete Liste; ohne dynam. Variablen realisiert *)
FROM InOut IMPORT (* PROC *) WriteCard, WriteLn;
CONST MaxLng = 100; (* Platz fuer maximal 100 Elemente
                      in der Kette *)

    PNIL      = 0;                                (* Pseudo-NIL *)
    Anfang    = PNIL + 1;                        (* Anfangsindex der Halde *)
    Ende      = Anfang + MaxLng - 1;             (* Endindex der Halde *)
TYPE ZeigTyp = [PNIL..Ende];                    (* Pseudo-Zeiger *)
    ZeigRec = RECORD
        Key : CARDINAL;
        Next: ZeigTyp;
        (* Cont: . *)
    END (* RECORD *);
VAR  Halde : ARRAY [Anfang..Ende] OF ZeigRec;    (* Ps.-Halde *)
    Anker, LeerListe : ZeigTyp;

PROCEDURE ZeigKette;                                (* Kette (global) ausgeben *)
VAR Hilfl : ZeigTyp;
BEGIN (* ZeigKette *)
    Hilfl := Anker;
    WHILE Hilfl # PNIL DO
        WriteCard (Halde[Hilfl].Key, 3);

```



```

        Hilfl := Halde[Hilfl].Next;
    END (* WHILE *);    WriteLn;
END ZeigKette;

PROCEDURE InitLeer;          (* Initialisierung der Leerliste *)
VAR i : ZeigTyp;
BEGIN (* InitLeer *)
    FOR i := Anfang TO Ende-1 DO Halde[i].Next := i+1; END;
    Halde[Ende].Next := PNIL; LeerListe := Anfang;
END InitLeer;

PROCEDURE New (VAR Neu: ZeigTyp);
BEGIN (* New *)
    Neu := LeerListe;      (* Elem. aus LeerListe nehmen,
                           wenn vorh. *)

    IF LeerListe # PNIL THEN
        LeerListe := Halde[LeerListe].Next;
    END (* IF *);
END New;

PROCEDURE Dispose (Alt: ZeigTyp);
BEGIN (* Dispose *)
    Halde[Alt].Next := LeerListe;
    LeerListe := Alt;      (* Alt an Anfang der LeerListe setzen *)
END Dispose;

PROCEDURE BaueKette (Lng: CARDINAL);
(* Aufbau der Kette von hinten nach vorn *)
(* Speicherueberlauf (New liefert PNIL) ist nicht
   abgefangen *)
VAR Hilfl : ZeigTyp;    i : CARDINAL;
BEGIN (* BaueKette *)
    Anker := PNIL; (* Kette initialisieren *)
    FOR i := Lng TO 1 BY -1 DO      (* rueckwaerts wegen Key *)
        New (Hilfl);      (* Neues Element erzeugen und besetzen *)
        WITH Halde[Hilfl] DO Key := i; Next := Anker; END;
        Anker := Hilfl;      (* Neues Elem.wird vor die Kette
                               gehaengt *)
    END (* FOR *);
END BaueKette;

BEGIN (* KetteMitFeld *)
    InitLeer;    BaueKette (9);    ZeigKette;
END KetteMitFeld.

```

2.4.5 Dateien (Files)

Bis jetzt waren wir in der Ein-/Ausgabe von Daten stark eingeschränkt. Die *Eingabe* erfolgte über die Tastatur oder durch Konstantendeklarationen, das Ziel von *Ausgaben* war ausschließlich der Bildschirm. Oft wäre es jedoch vorteilhaft gewesen, wenn wir eine Möglichkeit gehabt hätten, die Daten längerfristig im Rechner aufzubewahren.

Auch in der Praxis ist es meist nötig, große Datenmengen *permanent* zu speichern. Dazu gibt es in den Rechnersystemen viele unterschiedliche Medien außerhalb des Hauptspeichers, beispielsweise Magnetbänder, Harddisks oder Disketten. Durch ihre Vielfalt und ihre starke Abhängigkeit von Hardware-Charakteristika läßt sich der Zugriff auf solche *peripheren* oder externen Speicher jedoch nicht *direkt* in eine (Hardware-unabhängige) Programmiersprache einbinden. Vollends *inkompatibel* werden die Merkmale, wenn man zu den Speichern auch die Geräte für interaktive Ein- und Ausgabe, also insbesondere Bildschirme, Tastaturen und "pointing devices" (wie eine Maus) hinzunimmt.

In seiner Allgemeinheit ist dieses Problem bis heute nicht befriedigend gelöst. Ein wichtiger Sonderfall, die (sequentielle) Datei (File), konnte aber soweit standardisiert werden, daß eine (weitgehend) rechner- und geräteunabhängige Lösung möglich wurde. Solche Dateien sollen in diesem Abschnitt kurz behandelt werden. Komfortablere (weil nicht auf den sequentiellen Zugriff beschränkte) Datenbankschnittstellen sind auch zunehmend normiert worden, können aber wegen ihrer Komplexität hier nicht behandelt werden.

2.4.5.1 Eigenschaften und formale Beschreibung

Alle bisherigen Datentypen haben einen endlichen Wertebereich (also eine endliche Kardinalität), der Datentyp File hingegen ist prinzipiell von unendlicher Kardinalität. Mathematisches Strukturprinzip von Files ist die *Folgenbildung*. Rekursiv lassen sich Files als Folgen über einem Grundtyp wie folgt definieren:

Sei T ein Datentyp (*Grundtyp*).

Dann ist ϵ (*leere Folge*: Folge, die kein Element enthält) eine (T-) *Folge*.

Die Verkettung einer (T-) Folge mit einem Wert aus T ist wieder eine (T-) Folge.

- Notation:*
- (1) $\langle \rangle$ (oder ϵ) leere Folge.
 - (2) $\langle x_0 \rangle$ einelementige Folge.

Seien $x = \langle x_1, \dots, x_n \rangle$ und $y = \langle y_1, \dots, y_m \rangle$ Folgen eines beliebigen Grundtyps.

- (3) $x \circ y = \langle x_1, \dots, x_n, y_1, \dots, y_m \rangle$ Verkettung zweier Folgen.
- (4) $\text{first}(x) = x_1$ Bestimmung des 1. Folgelements falls $x \neq \epsilon$, ist $\text{first}(x)$ undefiniert.
- (5) $\text{rest}(x) = \langle x_2, \dots, x_n \rangle$ Bestimmung der Restfolge, der Folge ohne ihr 1. Element.

Aus (4) und (5) folgt: $x = \langle \text{first}(x) \rangle \circ \text{rest}(x)$.

Eine Datei kann man sich also vorstellen als eine - prinzipiell *beliebig* lange - Kette von Variablen des gleichen Typs (ähnlich einem Feld). Anders als bei einem Feld kann aber jeweils nur auf *eine* der Variablen zugegriffen werden, so als ob die Kette nur durch ein Fenster sichtbar wäre, das jeweils den Blick auf ein einziges Glied freigibt. Das Fenster ist die (zum Programm gehörende) *File-Variable*, das sichtbare Glied ist durch einen unsichtbaren *Schreib-/Lesezeiger* markiert.

Gegenüber der Lese-Schreibtechnik einer Turingmaschine ist das Verarbeitungsmodell bei Files sogar noch eingeschränkt, denn es sind nur folgende Operationen möglich:

- Eine (existierende oder neue) Datei wird mit einer im Programm *deklarierten* (File-)Variablen verbunden.
- Der Schreib-/Lesezeiger der Datei wird auf den *Anfang* gesetzt.
- Ein Element der Datei (also ein Glied der Kette) wird *gelesen*: dabei wird der Schreib-/Lesezeiger um ein Feld weiterbewegt.
- Ein Element der Datei wird *geschrieben*: dieses Element bildet anschließend das Ende der Datei, ein eventuell vorher vorhandener Rest der Datei wird dabei zerstört.

In PASCAL sind solche (ähnliche) Operationen Teil der Sprache, während sie in MODULA-2 über nichtnormierte (und damit implementierungsabhängige) Module bereitgestellt werden. Aus diesem Grund wird in den folgenden beiden Abschnitten die Dateiverarbeitung in PASCAL ausführlicher als die in MODULA-2 beschrieben.

Bei der interaktiven Ein- und Ausgabe ist eine für den Menschen lesbare Form unbedingt notwendig. Dazu werden die im Rechner gespeicherten Bitmuster entsprechend ihrer Bedeutung *interpretiert* (formatiert), also beispielsweise als ganze Zahl, als Dezimalbruch oder als Zeichenreihe. Das externe Medium (etwa der Drucker) zeigt in jedem Fall einen Text. Man spricht daher von einem *Textfile*. Die Elemente eines Textfiles sind also einzelne Zeichen (Objekte des Typs CHAR), die durch spezielle

Zeichen wie END OF LINE oder CARRIAGE RETURN strukturiert sein können. Prinzipiell ist diese Struktur aber von untergeordneter Bedeutung.

Für die Speicherung auf einem Datenträger ist diese Formatierung nicht notwendig, vielmehr ist eine *unformatierte* Speicherung bezüglich Übertragungszeit und Speicherbedarf effizienter. Darum gibt es neben der formatierten Ein-/Ausgabe auch die Möglichkeit der unformatierten Ein-/Ausgabe.

2.4.5.2 Dateien in PASCAL

In diesem Abschnitt soll im Zusammenhang die Behandlung von Dateien in der Sprache PASCAL vorgestellt werden: Deklaration, Operationen und ein Anwendungsbeispiel.

Deklaration von Files

type fT = File of T;

Jeder File vom Typ fT ist eine Folge von Null oder mehr Elementen vom Typ T.

Beispiele für Files:

type messreihe = file of int;

type stapel = file of Spielkarte;

Puffervariable

Mit der Deklaration eines Files f (*type f = ...*) ist implizit die Deklaration einer *Puffervariablen* $f^\uparrow$ verbunden. $f^\uparrow$ ist vom Typ, der als Grundtyp des Files vereinbart wurde.

$f^\uparrow$ wird i.a. als "Fenster" auf das File benutzt, bestimmt das jeweils *zugreifbare* File-Element (Position des Lese-Schreib-Kopfes) und spaltet das File (jeweils relativ zu $f^\uparrow$) in einen linken Teil f_L und einen rechten Teil f_R , d.h. $f = f_L \circ f_R$ mit $f^\uparrow = \text{first}(f_R)$.

Zugriffsmodi

Files erlauben nur *sequentiellen Zugriff*, d.h. nur Zugriff auf das jeweils nächste oder (nach "Rücklauf") auf das erste Element des Files. Beim schrittweisen Durchlaufen eines Files werden Lesen und Schreiben strikt getrennt.

Operationen auf Files

Elementare File-Operationen

Seien durch $\text{var } x : \text{file of } T_0$ eine File-Variable f und die Puffervariable $f\uparrow$ (vom Typ T_0) definiert.

- *Öffnen* eines Files
 $\text{open } (f)$ Effekt: File wird geöffnet (für weitere Operationen zugänglich gemacht)
 - *Vereinbaren* eines *leeren* Files bzw. *Löschen* eines bestehenden Files
 $\text{rewrite } (f)$ Effekt: $f := < >$;
 - *Erweitern* eines Files
 $\text{put } (f)$ Effekt: $f := f \circ f\uparrow$;
 - *Initialisieren* des File-Durchlaufs
 $\text{reset } (f)$ Effekt: $f_L := < >$;
 $f_R := f$;
 $f\uparrow := \text{first } (f_R)$;
 - *Fortsetzen* des File-Durchlaufs
 $\text{get } (f)$ Effekt: $f_L := f_L \circ \text{first } (f_R)$;
 $f_R := \text{rest } (f_R)$;
 $f\uparrow := \text{first } (f_R)$;
 - *Testen auf End of File-Prädikat*
 $\text{eof } (f)$ Effekt: $\text{eof } (f) = (f_R = < >)$;
- $\text{eof } (f)$ erhält also den Wert TRUE, wenn das Ende des Files bereits überschritten ist. Konsequenz: $f\uparrow$ ist undefiniert, weil $f_R = < >$! Einsichtiger wäre die Konvention $\text{eof } (f) = (\text{rest } (f_R) = < >)$. Dadurch bliebe $f\uparrow$ definiert, aber Standard-Algorithmen auf Files würden umständlicher.
- *Schließen* eines Files
 $\text{close } (f)$ Effekt: File wird geschlossen (und ist für weitere Operationen erst nach erneutem Öffnen wieder zugänglich)

Zusammengesetzte File-Operationen

Seien $\text{var } f: \text{file of } T_0$ und $\text{var } v: T_0$.

- **Beschreiben** des zugreifbaren File-Elementes
 $\text{write}(f, v)$ steht für $f \uparrow := v; \text{put}(f);$

- **Lesen** des zugreifbaren File-Elementes
 $\text{read}(f, v)$ Effekt: $v := f \uparrow; \text{get}(f);$

Voraussetzung: $\text{not eof}(f)$ vor Ausführung des read , damit kein undefiniertes Element gelesen wird. Beim letzten Durchlauf wird $f \uparrow$ undefiniert!

- **Schreiben** eines Files
 $\text{rewrite}(f);$
 $\text{while } p \text{ do}$
 $\text{begin } A(v); \text{write}(f, v)$
 $\text{end};$

- **Lesen** eines Files
 $\text{reset}(f);$
 $\text{while not eof}(f) \text{ do}$
 $\text{begin read}(f, v); A'(v)$
 $\text{end};$

Anwendungsbeispiel: "Mischen zweier Files"

Das Beispiel "Mischen zweier Files" soll anhand folgender Schritte erläutert werden:

- Analyse der Problemstellung
- Entwurf einer Lösung
- Codierung.

Problemstellung

Gegeben seien zwei Files f und g mit integer-Werten, die aufsteigend sortiert sind. Man "mische" f und g zu einem aufsteigend sortierten File h .

- a) **Analyse der Problemstellung**
 Gegeben:

f	...	f_{i-1}	f_i	f_{i+1}	...
$f_i \leq f_{i+1}$ für alle $i \in \{1, \dots, l_f - 1\}$					

g	...	g_{j-1}	g_j	g_{j+1}	...
$g_j \leq g_{j+1}$ für alle $j \in \{1, \dots, l_g - 1\}$					

Ziel:

h	...	h_{k-1}	k_k	h_{k+1}	...
---	-----	-----------	-------	-----------	-----

- mit 1. $h_k \leq h_{k+1}$ für alle $k \in \{1, \dots, l_f + l_g - 1\}$
 2. Alle Elemente von f und alle von g sind in h vorhanden.

b) *Entwurf einer Lösung (PASCAL-unabhängige Formulierung!)*

- I. Vereinbarung eines leeren Files h;
 Initialisierung der File-Durchläufe für f und g.
- II. Solange f oder g noch nicht ganz gelesen sind:
 wenn $f \uparrow$ kleiner ist als $g \uparrow$,
 dann kopiere Inhalt von $f \uparrow$ nach $h \uparrow$,
 sonst kopiere Inhalt von $g \uparrow$ nach $h \uparrow$.
 Erweitere bisheriges h um $h \uparrow$.
- III. Wenn f ganz gelesen ist,
 dann solange g noch nicht ganz gelesen ist:
 kopiere Inhalt von g nach $h \uparrow$; erweitere h um $h \uparrow$,
 sonst (d.h. wenn g ganz gelesen ist) solange f noch nicht ganz gelesen ist:
 kopiere Inhalt von f nach $h \uparrow$, erweitere h um $h \uparrow$.

c) *Codierung der Lösung (als PASCAL-Programm)*

```

procedure file (f,g,h);                                     (* Programm P2.51 *)
begin
  rewrite (h);
  reset (f); reset (g);
  while not eof (f) and not eof (g) do
    begin
      if  $f \uparrow < g \uparrow$  then
        read (f, $h \uparrow$ )
      else read (g, $h \uparrow$ );
      put (h)
    end;
  if eof (f) then
    begin
      while not eof (g) do
        begin
          read (g, $h \uparrow$ ); put (h)
        end
    end
  end
end

```

```

else
  begin
    while not eof (f) do
      begin
        read (f,h↑); put (h↑)
      end
    end
  end
end proc;

```

2.4.5.3 Dateien in MODULA-2

Wegen der unvermeidlichen Systemabhängigkeit wurde in MODULA-2 (anders als z.B. in PASCAL) folgendes Datei-Konzept gewählt: Die Dateien sind nicht Teil der Sprache, sondern werden von Modulen, die in der Regel als Teil des MODULA-2-Systems vorgegeben sind, bereitgestellt. Die Dateien werden also "gekapselt", d.h. vor den sie nutzenden Prozeduren und Funktionen "verborgen" (vgl. auch Kapitel 3 zum Thema "Abstraktion"). Leider ist der Inhalt solcher *kapselnder Module* für MODULA-2 nicht streng definiert, da unterschiedliche Systeme möglicherweise unterschiedliche Konzepte des Dateizugriffs verfolgen. Dieser Mangel macht MODULA-2-Programme unnötig systemabhängig. Die "offizielle" MODULA-2-Beschreibung von Wirth legt aber gewisse Eigenschaften dieser Module nahe.

Als primitives Basiskonzept dieser Module dient die sequentielle Datei, der sogenannte *Stream*. Die Prozeduren zur Verwaltung sequentieller Dateien werden aus dem Standard-Modul FileSystem importiert. Dieses stellt im strengen Sinne kein kapselndes Modul dar, weil die *interne* Struktur eines Files zugänglich ist. Dies macht zwar die Zugriffe effizienter und erlaubt in Sonderfällen das "Hineingreifen", man sollte davon aber wirklich nur in Sonderfällen Gebrauch machen und nur die angegebenen Standard-Prozeduren nutzen.

Für die *formatierte* Standard-Ein- /Ausgabe (typisch über Tastatur und Bildschirm) sind viele der vom Modul FileSystem angebotenen Operationen unnötig und komplizieren den Gebrauch. Auch führt die Behandlung der interaktiven Eingabe als Dateieingabe u.U. zu einem schwer verständlichen Verhalten des Systems. Darum sind spezielle Module (InOut und Terminal) für diesen Zweck geschaffen. InOut kapselt zwei Streams, die Textfiles in und out.

Als *Beispiel* für die Verwendung von Dateien in MODULA-2 ist nachfolgend ein Programm gezeigt, das Zahlen von einem Textfile einliest und in einem anderen File unformatiert speichert, bevor die Zahlen schließlich wieder eingelesen und formatiert ausgegeben werden. In der Praxis wird die Datei i.a. von verschiedenen Programmen verwendet.

Im Beispiel wird der Datentyp BYTE verwendet. Dieser Datentyp repräsentiert die kleinste adressierbare Speichereinheit (i.a. 8 bit). Einem formalen Parameter vom Typ BYTE kann jeder Ausdruck zugewiesen werden, dessen Wert einen gleichgroßen Speicherplatz belegt.

```

MODULE UnformEA;                                (* Programm P2.52 *)
(* Beispiel zur formatierten und unformatierten Ein-/Ausgabe in
   MODULA-2. Achtung, das Beispiel enthaelt einen beabsichtig-
   ten Typfehler (Ausgabe als Integer, Eingabe als Cardinal),
   vom Uebersetzer nicht erkannt werden kann. In realen Anwen-
   dungen sollten Typen bei Ein- und Ausgabe natuerlich gleich
   sein.
   Diese Implementierung ist ausserdem stark systemabhaengig. *)
FROM FileSystem IMPORT (* PROC *) Close, Lookup, File,
                        ReadChar, WriteChar;
FROM SYSTEM          IMPORT (* TYPE *) BYTE;
FROM InOut           IMPORT (* PROC *) ReadInt, WriteString,
                        WriteCard, WriteLn;

CONST FeldLng = 10;
TYPE  FeldIT  = ARRAY [1..FeldLng] OF INTEGER;
      FeldCT  = ARRAY [1..FeldLng] OF CARDINAL;
VAR   UnformDatei : File;
      bool        : BOOLEAN;
      FeldC       : FeldCT; FeldI: FeldIT;
      i           : CARDINAL;

PROCEDURE ReadRecord (VAR structure: ARRAY OF BYTE);
(* liest beliebige Datenstruktur (structure) von der Datei
   UnformDatei *)
VAR i : INTEGER;
    char : CHAR;
BEGIN (* ReadRecord *)
  Lookup (UnformDatei, "Hilfsdatei", FALSE); (* oeffnen *)
  FOR i := 0 TO HIGH (structure) DO
    ReadChar (UnformDatei, char);  structure [i] := char;
  END (* FOR *);
  Close (UnformDatei);
END ReadRecord;

PROCEDURE WriteRecord (structure: ARRAY OF BYTE);
(* schreibt beliebige Datenstruktur auf UnformDatei *)
VAR i : INTEGER;
BEGIN (* WriteRecord *)
  Lookup (UnformDatei, "Hilfsdatei", TRUE); (* oeffnen *)
  FOR i := 0 TO HIGH (structure) DO
    WriteChar (UnformDatei, structure [i]);
  END (* FOR *);
  Close (UnformDatei);
END WriteRecord;

```



```
BEGIN (* UnformEA *)
  FOR i := 1 TO FeldLng DO      (* formatierte Eingabe *)
    WriteString ('Integer: >');
    ReadInt (FeldI [i]);
    WriteLn;
  END (* FOR *);
  WriteRecord (FeldI);          (* unformatierte Ausgabe *)
  ReadRecord (FeldC);           (* unformatierte Eingabe;
                                Achtung, Typ-Inkonsistenz *)
  FOR i := 1 TO FeldLng DO
    WriteCard (FeldC [i], 10);  (* formatierte Ausgabe *)
    WriteLn;
  END (* FOR *);
END UnformEA.
```

3. Abstraktion

3.1 Abstraktionskonzepte in Programmiersprachen

Gegen Ende der 40er Jahre stand mit der von-Neumann-Architektur (siehe Abschnitt 1.3.1) ein universelles, durch Gleichbehandlung der Daten und Instruktionen homogenes Konzept zur Verfügung, das modernen Rechnern heute noch zugrundeliegt. Damals gab es noch keine eigentlichen Programmiersprachen; die Steuerung der Abläufe, die Eingabe und Ausgabe erfolgten auf sehr primitive, durch die *Hardware* bestimmte Art und Weise.

Aber sehr rasch zeigte sich, daß für die Formulierung der Algorithmen spezielle Notationen zweckmäßig sind, und es begann die Entwicklung der Programmiersprachen.

Nach gut vierzig Jahren können wir als durchgängige Idee bei dieser Entwicklung die Tendenz erkennen, den Programmierern immer höhere *Abstraktionsmechanismen* zur Verfügung zu stellen, so daß sich die Darstellungsebene immer weiter von der (physikalischen) Hardware hin zu der (*logischen*) Betrachtungsebene des Programmierers verschob.

Der Weg zu den höheren Programmiersprachen

Der von-Neumann-Rechner wurde zunächst *binär* programmiert, wie der folgende (fiktive) Programmausschnitt zeigt:

```
0000 1010 0000 0000 0011 1100 1000 1010
0000 0011 0000 0000 0000 0000 0000 0001
0000 1011 0000 0000 0011 1100 1000 1010
```

Durch die Abstraktion von einzelnen Bits entstand daraus ein *Oktal-* oder *Sedezimal-code*:

```
0A00 3C8A
0300 0001
0B00 3C8A
```

Die darin enthaltenen Befehlscodes wurden durch *Operator-Symbole* (mnemonische, d.h. sinnfällige, bedeutungstragende Codes vor den Adressen) weiter vereinfacht:

```
LDA 3C8A
AAD 0001
SPA 3C8A
```

Eine ähnliche Abstraktion setzte anstelle der Speicheradressen symbolische *Marken* und *Bezeichner*:

```
MARK1 =   LDA  ARGUM
          ADD  1
          SPA  ARGUM
```

Damit war die Ebene der *Assembler-Sprachen* erreicht.

Eine weitere wichtige Abstraktion bestand darin, die Formulierung von *Ausdrücken* direkt im Programm zuzulassen, das von einem Übersetzer durch "Formula Translation" (wie in der Programmiersprache FORTRAN erstmals sehr deutlich wurde) in den *Maschinencode* übertragen wurde:

```
ARGUM = ARGUM + 1
```

Die folgenden Schritte sind in Kapitel 2 bereits beschrieben: Durch die Abstraktion von Berechnungsvorschriften und ihren speziellen Daten entstanden *Funktionen* und *Parameter*, wie das folgende Pascal-Programm zeigt:

```
FUNCTION Fakultaet (Arg : integer): integer;
BEGIN
  IF Argument = 0 THEN Fakultaet := 1
    ELSE Fakultaet := Arg * Fakultaet (Arg-1)
END;
```

In ähnlicher Weise wurden Befehlssequenzen zu *Prozeduren* zusammengefaßt.

Datenabstraktion

Damit war der Stand erreicht, wie er *vor* MODULA-2 durch die gängigen Programmiersprachen repräsentiert wurde: Auf der Seite der *Befehle* (der imperativen Programmierung) und der *Ausdrücke* (funktionale Programmierung) konnte Komplexes in einfachen Anweisungen (Prozedur- und Funktionsaufrufen) integriert werden. Auf diese Weise ließen sich auch sehr umfangreiche Programme so strukturieren, daß zum Verständnis ihrer Teile jeweils nur *überschaubare* Programmabschnitte betrachtet werden mußten.

Unbefriedigend war dagegen die Abstraktion der *Daten*. Bei den primitiven Datentypen (INTEGER, REAL, BOOLEAN, CHAR) ist in den Sprachen mit strenger Typbindung ("strong typing") die Abstraktion vorgegeben: So stehen beispielsweise für eine BOOLEAN-Variable nur bestimmte Operationen zur Verfügung (z.B. die Negation, die Konjunktion und die Disjunktion durch die Operatoren NOT, AND und OR), während andere, z.B. die arithmetischen Operationen, nicht zugänglich sind. Bestimmte Daten und datenähnliche Betriebsmittel wie die Ein- und Ausgabe können

nicht direkt manipuliert werden, sondern nur durch vorgegebene Anweisungen (wie Write usw.).

Diese Beschränkungen bringen zwei Vorteile: Zum einen kann der Übersetzer viele triviale Fehler anzeigen, die beim Codieren unterlaufen. Zum anderen können wir unsere Programme nicht auf spezielle Eigenschaften *unserer* Arbeitsumgebung zuschneiden, wenn diese Eigenschaften unsichtbar sind ("*information hiding*"). Damit tragen die Beschränkungen zur *Korrektheit* und *Portabilität* der Programme bei.

Dieser sinnvolle Schutz läßt sich auf Datenstrukturen, die vom Programmierer selbst geschaffen sind, nicht ohne weiteres übertragen, wie das folgende Beispiel zeigt:

Sei S ein Zeichenkette der Länge L, $0 \leq L \leq L_{\text{Max}}$. Wir wollen als Operationen auf S erlauben:

- die Reduktion auf eine bestimmte Länge L_1 ($0 \leq L_1 \leq L$)
- die Verlängerung um ein Zeichen Z
- die Ausgabe.

Wir realisieren S als Record:

```
S : RECORD
    SCont: ARRAY [0..LMax-1] OF CHAR; (* die Zeichenkette *)
    SL    : CARDINAL;                  (* ihre aktuelle Laenge *)
END (* RECORD *);
```

Jede Prozedur P, die auf S Zugriff hat, kann auch auf die Komponenten SCont und SL zugreifen und diese auf beliebige und damit auch auf unerwünschte Art verändern. P ist damit nicht nur "gefährlich" für S, sondern auch auf diese spezielle Realisierung von S zugeschnitten und verbietet eine Änderung, beispielsweise die Darstellung durch eine Zeichenreihe mit Ende-Markierung, also ohne *explizite* Speicherung der Länge.

Datenabstraktion besteht nun darin, die Vorteile der vordefinierten Datentypen für die neu geschaffenen zu erhalten. Dies geschieht durch das *Verbergen* der Variablen S in einem *Modul*, so daß nur die Operationen auf S von außen sichtbar sind, nicht S selbst ("*Kapselung*"). Soll dieses Konzept von speziellen Variablen auf einen Typ ausgedehnt werden, so wird der Typ ohne Informationen über seine Realisierung zur Verfügung gestellt ("*Abstrakter Datentyp*").

Abstraktionskonzepte in neueren Sprachen

In sogenannten *objektorientierten Sprachen* (z.B. Smalltalk-80 und C++) ist das oben skizzierte Konzept der Kapselung und der Abstrakten Datentypen noch einmal wesentlich erweitert worden. Die Operationen ("*Methoden*" genannt) sind hier syntaktisch und logisch den *Typen* der Objekte, den *Klassen*, zugeordnet. Die sogenannte

Vererbung, d.h. die Übernahme der Merkmale und Operationen von einer Klasse in ihre Unterklassen, ist ein weiteres für die objektorientierten Sprachen charakteristisches Abstraktionskonzept.

Die Programmiersprache Ada ist zwar keine objektorientierte Sprache, bietet aber in Ansätzen ähnliche Möglichkeiten, und zwar

- durch *Overloading*, d.h. durch die Verwendung desselben Bezeichners für verschiedene Operationen, die durch den Kontext unterschieden werden (einfaches Beispiel: “+” für ganze oder komplexe Zahlen),
- durch *Generic Units*, also Programmteile, die insgesamt parametrisiert werden können, z.B. durch Typen oder Funktionen (einfaches Beispiel: Generischer Listentyp, der z.B. für die Erzeugung von INTEGER- oder REAL-Listentypen verwendet werden kann).

3.2 Abstraktion in MODULA-2

In diesem Abschnitt werden zwei Prinzipien eingeführt, die der Abstraktion dienen und in MODULA-2 sehr elegant integriert sind, nämlich

- die Auftrennung des Programms in mehrere Übersetzungseinheiten (*Modularisierung*) und
- die *Kapselung* von Daten und Typen.

3.2.1 Das Prinzip der separaten Übersetzung

Das Prinzip der getrennten Übersetzung wurde zwar schon in der Programmiersprache FORTRAN realisiert, doch gibt es dort keine *Schnittstellenprüfung* der getrennten Programmeinheiten.

Zur Erläuterung dieses Mangels diene folgendes Beispiel:

Nehmen wir an, eine Prozedur erhält als Wertparameter eine positive ganze Zahl (CARDINAL), die in der Prozedur als Code eines Zeichens interpretiert und weiterverarbeitet wird. Der Programmierer beschließt nun, anstatt des Codes das Zeichen selbst zu übergeben (CHAR). Die Prozedur wird dementsprechend umgestaltet, nur leider wird die Anpassung des Aufrufs in einem anderen Modul (d.h. einer anderen *Übersetzungseinheit*) vergessen. Zur Laufzeit würde dann einer Prozedur, die ein Zeichen erwartet, eine Zahl übergeben. Das besonders unangenehme ist nun, daß die Reaktion des Hauptprogramms völlig *unberechenbar* ist. Möglich wäre zum Beispiel, daß ein Teil der übergebenen Zahl als einzelnes Zeichen interpretiert wird, so daß die Prozedur ein falsches Ergebnis liefert. Ein "Programmabsturz" wäre aber genauso möglich! Solche Fehler sind sehr schwer zu lokalisieren.

In MODULA-2 dagegen erfolgt bei der *separaten Übersetzung* eine Schnittstellenprüfung, und es wäre (um beim Beispiel zu bleiben) gar nicht möglich, daß einer Prozedur, die eine Zahl erwartet, ein Zeichen übergeben wird.

Schnittstelle und Implementierung

Ziel bei der Konzeption von MODULA-2 war es, die Module so unabhängig wie möglich zu machen. Insbesondere soll ein (*Kunden-*)Modul, das ein anderes (das *Dienst-*)Modul in Anspruch nimmt, nicht von den Details dessen Implementierung abhängen. Um trotzdem eine Schnittstellenprüfung zu erreichen, wird das Dienst-Modul in zwei Teile gegliedert (siehe Abb. 3.1):

- *Schnittstelle* ("Definition Module") und
- *Implementierung* ("Implementation Module").

Abb. 3.1 Gliederung in Schnittstelle und Implementierung

Die logische Verknüpfung von Modulen erfolgt durch *Importieren* der Bezeichner aus *Schnittstellen*.

Diese Lösung hat u.a. folgende *Vorteile*:

- Die Schnittstellenprüfung kann ohne Kenntnis der Implementierung erfolgen.
- Eine Neuimplementierung der in der Schnittstelle deklarierten Prozeduren, bei der die Funktion unverändert bleibt, zieht keine Änderung des importierenden Moduls nach sich, auch keine Neuübersetzung.
- Bei den Prozeduren ist eine höhere Abstraktion erreicht, denn ihre Realisierung ist unsichtbar.
- Die Bearbeitung und Übersetzung ist jeweils auf relativ kleine, überschaubare und leicht zu handhabende Einheiten begrenzt.

Diesen Vorteilen steht als *Nachteil* gegenüber, daß die Konsistenz schwerer zu erreichen ist als ohne Modularisierung, weil Unklarheiten nicht anhand der Implementierung beseitigt werden können.

Daher sind besonders wichtig:

- aussagekräftige Bezeichner
- ausführliche Kommentare in den Schnittstellen, die die Funktion der Prozeduren beschreiben, auch für Fehler- und Sonderfälle.

Die *Syntax* für die Schnittstellen- und Implementierungsdefinition in MODULA-2 läßt sich wie folgt beschreiben (für die vollständige Syntax siehe Syntaxdiagramme im Anhang B):

Schnittstelle = "DEFINITION" "MODULE" identifier ";"
 {import} {definition} "END" identifier ".".

definition = { "CONST" ConstDeclaration | "TYPE" TypeDefinition |
 "VAR" VariableDeclaration | ProcedureHeading }

Diese Festlegung der Syntax für Definitionen bedeutet, daß die Schnittstelle nur Konstanten- und Typdefinitionen, Variablendeklarationen und Prozedurköpfe enthalten darf, aber keine ausführbaren Anweisungen!

Implementierung = "IMPLEMENTATION" "MODULE" identifier ";"
 {import} block identifier ".".

Der Anweisungsblock am Ende ist leer (d.h. auf END reduziert) oder enthält *Initialisierungsoperationen*, vor allem für lokale Variablen.

Eine MODULA-2-System behandelt die Übersetzungseinheiten folgendermaßen:

- *Übersetzung:* Schnittstelle → Symboltabelle mit Zeitmarken
- *Übersetzung:* Implementierung → Objectcode
 Der Objectcode enthält die Zeitmarken aller in Anspruch genommenen Schnittstellen. Dazu gehört stets die *eigene* Schnittstelle.
- *Binden:* Objectcode → ausführbares Programm
 Der Binder (Linker) prüft die Konsistenz der Zeitmarken (die von einer bestimmten Schnittstelle stammende Zeitmarke muß im ganzen System einheitlich sein!).
- *Laden:* ausführbares Programm → Programmlauf, Ergebnisse
 Bei einigen Systemen ist der Linker im Laufzeitsystem enthalten, das Binden findet also im Standardfall bei Programmausführung implizit statt.

3.2.2 Modularisierung eines Programms

Jedes praxisrelevante, d.h. in größeren Softwareprojekten interessierende Programm besteht aus mehreren Modulen, von denen höchstens eines ein nicht in Schnittstelle und Implementierung gegliederter *Programmodul* sein kann. Dieses ist das *Hauptprogramm*.

Beim *Entwurf* der Module, d.h. der Modularisierung des Programms, sind folgende Gesichtspunkte zu beachten:

- Die Verwendung von Objekten aus anderen Modulen in einer Schnittstelle erfordert deren *Import*. Deren Gültigkeit beschränkt sich allerdings auf die Schnittstelle selbst. Werden diese Objekte auch in der Implementierung benutzt, so müssen sie dort erneut importiert werden (z.B. der Typ ADDRESS aus SYSTEM: Er wird manchmal bei Prozedurparametern benötigt und muß daher in den Importlisten der Schnittstelle *und* der Implementierung erscheinen).
- Die erste Regel gilt nur für Importe aus *anderen* Modulen, nicht für Konstanten, Typen und Variablen, die in der zugehörigen Schnittstelle deklariert sind. Diese dürfen *nicht* in der Implementierung desselben Moduls erneut deklariert werden, denn sie sind implizit importiert. Nur die Prozedurköpfe werden wiederholt.
- Ein *zyklischer* Import (Beispiel: die Schnittstelle eines Moduls M2 importiert aus Modul M1 und umgekehrt) zwischen Schnittstellen ist nicht möglich. Wohl erlaubt ist hingegen, daß die Implementierungen gegenseitig aus den Schnittstellen importieren.

Die Anweisungsblöcke der Module werden, soweit sie nicht leer sind, in einer durch den sogenannten *Import-Graphen* (siehe Abb. 3.2) bestimmten Reihenfolge ausgeführt: Wenn ein Kunden-Modul M aus M1 importiert, so wird zuerst M1 ausgeführt. Importiert M aus M1 und M2 oder importieren sich M1 und M2 gegenseitig, so ist die Ausführungsreihenfolge von M1 und M2 unbestimmt.

Falsch:

Richtig:

("A → B" heißt: "Modul B importiert aus A"; gestrichelt: impliziter Import)

Abb. 3.2 Falscher und richtiger Import-Graph

Bei der Verwendung der Module ist weiter zu beachten:

- Nur Objekte, die in der Schnittstelle *deklariert* (nicht importiert!) sind, werden *exportiert*.
In älteren MODULA-2-Systemen wurden die zu exportierenden Objekte explizit in einer *Export-Liste* angegeben; jetzt gibt es diese Liste nur noch bei den (hier nicht behandelten) geschachtelten Modulen.
- Beim Import von Aufzählungstypen, z.B. des Typs AT mit der Definition "AT = (u,v,w);", werden die Konstanten (hier also u,v,w) *implizit* importiert. Das gleiche gilt für die Selektoren bei Recordtypen.
- Der Basistyp eines importierten Sets wird dagegen nicht automatisch importiert (im folgenden Beispiel also Element zu SetTyp):

```
Element= (gelb, gruen, rot); SetTyp = SET OF Element;
```

Die Importliste kann folgendermaßen aussehen:

- Import einzelner Objekte:

```
FROM Modul1 IMPORT a,b,c;
(* a,b und c müssen in der Schnittstelle
   von Modul1 deklariert sein *)
```

Verwendung:

```
...; a := b + c; ....
```

- Import des ganzen Moduls:

```
IMPORT Modul1;
```

Verwendung einzelner Objekte:

```
...; Modul1.a := Modul1.b + Modul1.c; ...
```

Beispiel: Modularisierung von DoppelKette

Wir können Programm P2.48 (Modul DoppelKette) z.B. wie folgt in mehrere Module aufteilen und für ein Kunden-Modul zugänglich machen:

Schnittstelle

```
DEFINITION MODULE EinfuegenUndLoeschen;    (* Programm P3.1a *)
(* Modifizierte, gegliederte Form des Programms P2.48 *)
TYPE ZeigTyp = POINTER TO ZeigRec;
    ZeigRec = RECORD
        Key          : CARDINAL;
        Prev, Next   : ZeigTyp;
    END (* RECORD *);
VAR Anker : ZeigTyp;
```



```

BEGIN (* EinfuegenUndLoeschen *)
  (* Kette mit fliegendem Anker initialisieren *)
  ALLOCATE (Anker, SIZE (ZeigRec));
  WITH Anker^ DO
    Key := 0;   Next := Anker;   Prev := Anker;
  END (* WITH *);
END EinfuegenUndLoeschen.

```

Test-Modul

```

MODULE DoppelKette;                                (* Programm P3.1c *)
FROM InOut                                          IMPORT (* PROC *) WriteCard, WriteLn;
FROM EinfuegenUndLoeschen IMPORT (* TYPE *) ZeigTyp, ZeigRec,
                                (* VAR *) Anker,
                                (* PROC *) Einfuegen, Loeschen;

PROCEDURE ZeigKette;
VAR Hilfl : ZeigTyp;
BEGIN (* ZeigKette *)
  Hilfl := Anker^.Next;
  WHILE Hilfl # Anker DO
    WriteCard (Hilfl^.Key, 3);
    Hilfl := Hilfl^.Next;
  END (* WHILE *);
  WriteLn;
END ZeigKette;

BEGIN (* DoppelKette *)
  Einfuegen(7);   Einfuegen(1);   Einfuegen(4);
  Einfuegen(5);   Einfuegen(9);
  Loeschen (1);   Loeschen (9);   Loeschen (5);   ZeigKette;
END DoppelKette.

```

Ein Problem bleibt bei dieser Art der Modularisierung noch ungelöst: Wir haben zwar von den Prozeduren, aber nicht von den Daten abstrahiert. Unsere Forderung lautet darum: Es sollen keine Daten und Typen mehr exportiert werden, es sei denn, sie dienen zur Kommunikation (als Parameter). Damit erreichen wir die Ebene der *Datenkapselung*.

3.2.3 Datenkapselung

Das Prinzip der vollständigen Datenkapselung läßt sich wie in Abb. 3.3 dargestellt grafisch veranschaulichen. Wir sehen, daß bei dieser vollständigen Kapselung keine Variable exportiert werden darf.

Abb. 3.3 Datenkapselung

Beispiel: Datenkapselung

Für das durchgängige Beispiel der DoppelKette muß die Prozedur **ZeigKette** in das kapselnde Modul "wandern". Es entstehen folgende Übersetzungseinheiten:

Schnittstelle

```

DEFINITION MODULE KettenKapsel;                                (* Programm P3.2a *)
(* Besser strukturierte Variante des Programms P3.1 *)

PROCEDURE Einfuegen (Nr: CARDINAL);
(* fuegt neues Element in die geordnete Liste ein. *)

PROCEDURE Loeschen (Nr: CARDINAL);
(* loescht Elem. in Liste.
   Element mit Key = Nr muss vorhanden sein *)

PROCEDURE ZeigKette;
(* zeigt den aktuellen Stand der Kette auf dem Bildschirm *)

END KettenKapsel.
  
```

Implementierung

```

IMPLEMENTATION MODULE KettenKapsel;                            (* Programm P3.2b *)
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
FROM InOut  IMPORT (* PROC *) WriteCard, WriteLn;
TYPE ZeigTyp = POINTER TO ZeigRec;
TYPE ZeigRec = RECORD
    Key          : CARDINAL;
    Prev, Next  : ZeigTyp;
END (* RECORD *);
  
```

```

VAR Anker    : ZeigTyp;

PROCEDURE Einfuegen (Nr: CARDINAL);
  ... wie oben in Programm P3.1b ...
END Einfuegen;

PROCEDURE Loeschen (Nr: CARDINAL);
  ... wie oben in Programm P3.1b ...
END Loeschen;

PROCEDURE ZeigKette;
  ... wie oben in Programm P3.1c ...
END ZeigKette;

BEGIN (* KettenKapsel *)
  ... Initialisierung wie oben in Programm P3.1b ...
END KettenKapsel.

```

Test-Modul für Kettenkapsel

```

MODULE DoppelKette;                                     (* Programm P3.2c *)
(* Version mit Kapselung der Daten *)
FROM KettenKapsel IMPORT (* PROC *) Einfuegen, Loeschen,
                           ZeigKette;

BEGIN (* DoppelKette *)
  ... Hauptprogramm wie in Programm P3.1c ...
END DoppelKette.

```

ZeigTyp, ZeigRec und der Anker sind nun *private* Informationen der Implementierung, Informationen also, die anderen Modulen nicht bekannt sind und daher von diesen auch nicht *mißbraucht* werden können (hier geht es um den fahrlässigen, nicht den vorsätzlichen Mißbrauch!). Dieses Prinzip wurde 1972 von David Parnas unter der Bezeichnung "*information hiding*" eingeführt.

Abb. 3.4 Unkontrollierter und kontrollierter Datenzugriff

Abb. 3.4 stellt dem unkontrollierten, freien Datenzugriff die Lösung mit Kapselung, d.h. den kontrollierten Zugriff über Prozeduren gegenüber.

Die Kapselung hat folgende *Vorteile*: Die Datenstrukturen sind vor Fehlinterpretationen geschützt, Mißbrauch ist erschwert. Die gekapselte Datenstruktur kann ohne Auswirkung auf andere Module geändert werden. Ein *Nachteil* ist, daß die Datenzugriffe i.a. etwas weniger effizient sind.

Beispiel: Kettenkapsel mit Navigationsoperationen

Soll aus irgendeinem Grunde (der hier nicht gegeben ist) *ZeigKette* im Kunden-Modul bleiben, so müssen sogenannte *Navigationsoperationen* (hier *InitLesen* usw.) aus *KettenKapsel* exportiert werden, damit trotzdem eine vollständige Kapselung der Datenstruktur erreicht ist.

Schnittstelle

```

DEFINITION MODULE KettenKapsel2;                (* Programm P3.3a *)
(* Variante des Programms P3.2 mit Kapselung und Navigation *)

PROCEDURE Einfuegen (Nr: CARDINAL);
(* fuegt neues Element in die geordnete Liste ein. *)

PROCEDURE Loeschen (Nr: CARDINAL);
(* loescht Element mit Key = Nr in Liste; muss vorh. sein *)

PROCEDURE InitLesen;

PROCEDURE TestEnde (): BOOLEAN;

PROCEDURE LesenUndWeiter (VAR Schluessel: CARDINAL);
(* InitLesen bereitet Lesen der Kette vor, TestEnde zeigt an,
   ob das Ende erreicht ist. LesenUndWeiter liefert in
   Schluessel den Schluessel des anstehenden Elements; war
   zuvor TestEnde = FALSE, so wird der Lesezeiger weiterge-
   setzt. Ist die Kette leer, liefert TestEnde stets TRUE *)

END KettenKapsel2.

```

Implementierung

```

IMPLEMENTATION MODULE KettenKapsel2;            (* Programm P3.3b *)
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
TYPE ZeigTyp = POINTER TO ZeigRec;
   ZeigRec = RECORD
       Key          : CARDINAL;
       Prev, Next  : ZeigTyp;
   END (* RECORD *);
VAR Anker, LeseZeiger : ZeigTyp;

```



```

PROCEDURE Einfuegen (Nr: CARDINAL); wie oben in Programm
P3.1b

PROCEDURE Loeschen (Nr: CARDINAL); wie oben in Programm
P3.1b

PROCEDURE InitLesen;
BEGIN LeseZeiger := Anker^.Next; END InitLesen;

PROCEDURE TestEnde (): BOOLEAN;
BEGIN RETURN LeseZeiger = Anker; END TestEnde;

PROCEDURE LesenUndWeiter (VAR Schluessel: CARDINAL);
BEGIN (* LesenUndWeiter *)
  Schluessel := LeseZeiger^.Key;
  IF NOT TestEnde() THEN LeseZeiger := LeseZeiger^.Next; END;
END LesenUndWeiter;

BEGIN (* KettenKapsel2 *)
  (* Kette mit fliegendem Anker initialisieren *)
  ALLOCATE (Anker, SIZE (ZeigRec));
  WITH Anker^ DO Key := 0; Next := Anker; Prev := Anker; END;
  InitLesen; (* damit TestEnde definierten Wert liefert *)
END KettenKapsel2.

```

Test-Modul für Kettenkapsel2

```

MODULE DoppelKette2;                                     (* Programm P3.3c *)
(* Version mit Kapselung der Daten und Navigation zum Lesen *)
FROM InOut IMPORT (* PROC *) WriteCard, WriteLn;
FROM KettenKapsel2 IMPORT (* PROC *) Einfuegen, Loeschen,
                           InitLesen, TestEnde,
                           LesenUndWeiter;

PROCEDURE ZeigKette;
VAR Schluessel : CARDINAL;
    Abbruch    : BOOLEAN;
BEGIN (* ZeigKette *)
  InitLesen;
  WHILE NOT TestEnde () DO
    LesenUndWeiter (Schluessel);
    WriteCard (Schluessel, 3);
  END (* WHILE *);
  WriteLn;
END ZeigKette;

BEGIN (* DoppelKette2 *)
  ... Hauptprogramm wie in Programm P3.1c ...
END DoppelKette2.

```

Beispiel: Warteschlange

Als ein weiteres Beispiel folgt die vollständige Kapselung einer Warteschlange, also einer Datenstruktur, in der Informationen (hier: CARDINAL-Werte) nach dem sogenannten *FIFO-Prinzip* (First In First Out) gespeichert werden können.

Schnittstelle

```

DEFINITION MODULE SchlKapsel;                                (* Programm P3.4a *)
(* Demonstr. Kapselung: FIFO-Speicher fuer CARDINAL-Zahlen *)

PROCEDURE Bringen (Eintrag : CARDINAL);
(* wenn istVoll vorher TRUE war, keine Wirkung;
   sonst wird Eintrag der Schlange zugefuegt *)

PROCEDURE Holen (VAR Eintrag : CARDINAL);
(* wenn istLeer vorher TRUE war, keine Wirkung;
   sonst wird Par. Eintrag mit dem aeltesten Eintrag der
   Schlange besetzt, und dieser wird aus der Schlange
   entfernt *)

PROCEDURE istLeer (): BOOLEAN;
(* TRUE, genau wenn Schlange kein Element enthaelt *)

PROCEDURE istVoll (): BOOLEAN;
(* TRUE, genau wenn Schlange kein Element mehr
   aufnehmen kann *)

END SchlKapsel.
```

Implementierung

```

IMPLEMENTATION MODULE SchlKapsel;                            (* Programm P3.4b *)
(* Demonstr. der Kapselung: FIFO-Speicher fuer CARDINAL-Zahlen
   Version mit Ringpuffer *)
CONST MaxElemente = 100;
      Feldlaenge = MaxElemente+1;                            (* Minim. 1 Platz
                                                                bleibt frei *)
TYPE SchlIndTyp = [0 .. Feldlaenge-1];
VAR Schlange : ARRAY SchlIndTyp OF CARDINAL;
      AnfangVoll, AnfangLeer: SchlIndTyp;                    (* die gekapselte *)
                                                                (* Datenstruktur *)

PROCEDURE Nachfolger (Arg: SchlIndTyp) : SchlIndTyp;
(* intern ,zyklische Nachfolgefunktion *)
BEGIN (* Nachfolger *)
  RETURN (Arg + 1) MOD Feldlaenge;
END Nachfolger;
```

```

PROCEDURE Bringen (Eintrag : CARDINAL);
BEGIN (* Bringen *)
  IF NOT istVoll () THEN
    Schlange [AnfangLeer] := Eintrag;
    AnfangLeer := Nachfolger (AnfangLeer);
  END (* IF *);
END Bringen;

PROCEDURE Holen (VAR Eintrag : CARDINAL);
BEGIN (* Holen *)
  IF NOT istLeer () THEN
    Eintrag := Schlange [AnfangVoll];
    AnfangVoll := Nachfolger (AnfangVoll);
  END (* IF *);
END Holen;

PROCEDURE istLeer () : BOOLEAN;
BEGIN (* istLeer *)
  RETURN AnfangVoll = AnfangLeer;
END istLeer;

PROCEDURE istVoll () : BOOLEAN;
BEGIN (* istVoll *)
  RETURN Nachfolger (AnfangLeer) = AnfangVoll;
END istVoll;

BEGIN (* Initialisierung *)
  AnfangVoll := 0;
  AnfangLeer := 0;
END SchlKapsel.

```

Test-Modul für SchlKapsel

```

MODULE KapselTest;                                     (* Programm P3.4c *)
FROM SchlKapsel IMPORT (* PROC *)   Bringen, Holen,
                                istLeer, istVoll;
FROM InOut             IMPORT (* PROC *)   ReadCard, Write, WriteLn,
                                WriteCard, WriteString;

PROCEDURE WriteB (Value: BOOLEAN);
  (* Ausgabe von Wahrheitswerten *)
BEGIN (* WriteB *)
  IF Value THEN Write ('T'); ELSE Write ('F'); END;
END WriteB;

VAR Wert : CARDINAL;

BEGIN (* KapselTest *)
  WriteString ('*** Start Test-Programm f. SchlKapsel ***');
  WriteLn; WriteLn; WriteString ('Gib Zahl ohne Vorzeichen,');
  WriteString (' 0 fuer Ausgabe, 999 fuer Ende');
  WriteLn; WriteLn;

```



```

LOOP          (* jeweils Ein- oder Ausgabe eines Wertes *)
  Write ('>');  ReadCard (Wert);  Write (' ');
  IF Wert = 999 THEN EXIT;          (* Abbruchbedingung *)
  ELSIF Wert = 0 THEN                (* Wert aus Schlange holen *)
    IF istLeer () THEN
      WriteString ('  Schlange ist leer');
    ELSE
      Holen (Wert); WriteCard (Wert, 4);
      WriteString ('    istLeer = '); WriteB (istLeer ());
    END (* IF istLeer () *);
  ELSE                                     (* Wert in Schlange setzen *)
    IF istVoll () THEN
      WriteString ('  Schlange ist voll');
    ELSE
      Bringen (Wert);
      WriteString ('    istVoll = ');
      WriteB (istVoll ());
    END (* IF istVoll () *);
  END (* IF *);
  WriteLn;
END (* LOOP *);
WriteString ('*** Ende Test ***');  WriteLn;
END KapselTest.

```

3.2.4 Abstrakte Datentypen

3.2.4.1 Das Prinzip des Abstrakten Datentyps

Wie ein Typ das Schema ist, aus dem beliebig viele Variablen gebildet werden können, so ist der Abstrakte Datentyp (ADT) ein allgemeines Schema zur Bildung geschützter Variablen im Sinne der Kapselung. Anschaulich betrachtet ist ein ADT

- der *Name eines Typs*, dessen innere Struktur nicht sichtbar ist, und
- die *Menge der zulässigen Operationen* auf Objekten dieses Typs.

Es handelt sich also nicht wie bei der Kapselung um ein einzelnes Objekt, sondern um einen Typ, von dem es (durch Deklarationen in anderen Modulen, die den Typ importieren) beliebig viele Variablen geben kann.

Die *Initialisierung* muß nun für jedes Objekt einzeln erfolgen, erfordert also in jedem Fall eine aufrufbare Operation. Wo die Initialisierung eine Speicherbelegung auf der Halde impliziert (in MODULA-2 fast unvermeidlich), ist i.a. auch eine Lösch-Operation (zur Speicherfreigabe) nötig.

Graphisch läßt sich der ADT wie in der Abb. 3.5 dargestellt veranschaulichen.

Abb. 3.5 Abstrakter Datentyp

Abstrakte Datentypen lassen sich durch die nachfolgenden *Charakteristika* präzisieren.

- Einsatz:**
- Beliebig erweiterbare Menge gekapselter Daten eines bestimmten Typs.
- Prinzip:**
- Der Typ, auf den sich die Operationen des ADT beziehen, wird in der Schnittstelle deklariert, nicht aber definiert. Es handelt sich hierbei also um einen *opaken* oder *privaten Typ*.
 - Die Kunden-Module importieren den opaken Typ samt den Prozeduren und legen Variablen an.
 - Die Initialisierung der Variablen kann nicht implizit im exportierenden Modul erfolgen, sondern bedarf einer eigenen Prozedur.
 - *Alle* Operationen (also neben der Initialisierung z.B. auch Vergleich und Wertzuweisung) müssen durch das Modul ausgeführt werden, das den ADT exportiert. Die betreffenden Variablen werden als Parameter übergeben.
- Vorteile:**
- Durch die Kapselung der Datenstrukturen sind Fehlinterpretationen vermindert und ist Mißbrauch erschwert.
 - Die Datenstruktur kann ohne Auswirkungen auf andere Module geändert werden.
 - Die Zahl der Variablen, die aus einem ADT gebildet werden, bleibt offen. Diese Variablen können beispielsweise auch in rekursiven Prozeduren deklariert oder als Komponenten von Datenstrukturen verwendet werden.

- Nachteile:
- Die Datenzugriffe sind im allgemeinen etwas weniger effizient.
 - Durch die Art, wie ADTen in MODULA-2 realisiert werden, entstehen Risiken bei der naiven Verwendung (ein Beispiel folgt in Abschnitt 3.2.4.4).

Damit in einer Programmiersprache ADTen gebildet werden können, sollten vier *Forderungen* erfüllt sein:

- (1) Der Compiler muß in der Lage sein, Speicherplatz für die Variablen eines ADTs zu belegen, z.B. für K1, K2, F in

```
FROM KomplexLibrary IMPORT (* TYPE *) Komplex;
VAR K1, K2 : Komplex;
    F      : ARRAY [1 .. 100] of Komplex;
```

- (2) Die innere Struktur des Typs darf nicht eingeschränkt werden.
- (3) Die innere Struktur des Typs darf nicht für andere Module sichtbar werden; eine Änderung darf sich dort nicht auswirken.
- (4) Die Zugriffe auf Variablen eines ADTs sollten nicht wesentlich anders und etwa gleich effizient wie die auf Variablen anderer Typen sein.

Diese Forderungen lassen sich nicht miteinander vereinbaren. Bei MODULA-2 wurden Abstriche bei Punkt (4) und vor allem bei Punkt (2) gemacht.

Der Speicherbedarf des opaken Typs wurde auf die Länge eines Pointers festgelegt. Man ist also in aller Regel gezwungen, den opaken Typ als POINTER TO BasisTyp zu implementieren. Entsprechend können die Datenstrukturen nicht im *Kellerspeicher* liegen, die Zugriffe sind etwas aufwendiger, und wegen der Zeiger bieten Wert-Parameter eines ADTs keinen Schutz vor Veränderung durch das Unterprogramm (siehe Abschnitt 3.2.4.3).

3.2.4.2 Abstrakte Datentypen Schlange und Stack

Nachfolgend werden zwei oft verwendete Speicherstrukturen als ADTen eingeführt:

- ADT Schlange zur Realisierung einer *FIFO-Speicherstruktur*,
- ADT Stack für die Realisierung einer *LIFO-Speicherstruktur* (Last In First Out).

ADT Schlange

Das folgende Modul ADTSchlange stellt einen ADT für Schlangen von CARDINAL-Zahlen zur Verfügung.

Schnittstelle

```

DEFINITION MODULE ADTSchlange;                                (* Programm P3.5a *)
(* Demonstration eines ADTs:
   FIFO-Speicher fuer CARDINAL-Zahlen *)
TYPE SchlangenTyp;                                           (* ein privater (opaker) Typ *)

PROCEDURE Bringen (VAR Schlange : SchlangenTyp;
                  Eintrag  : CARDINAL);
(* wenn istVoll vorher TRUE war, keine Wirkung;
   sonst wird der Eintrag der Schlange zugefuegt *)

PROCEDURE Holen (VAR Schlange : SchlangenTyp;
                VAR Eintrag  : CARDINAL);
(* wenn istLeer vorher TRUE war, keine Wirkung; sonst wird
   Eintrag mit dem aeltesten Eintrag der Schlange besetzt,
   und dieser wird aus der Schlange entfernt *)

PROCEDURE istLeer (Schlange : SchlangenTyp): BOOLEAN;
(* TRUE, wenn Schlange kein Element enthaelt *)

PROCEDURE istVoll (Schlange : SchlangenTyp): BOOLEAN;
(* TRUE, wenn Schlange kein Element mehr aufnehmen kann *)

PROCEDURE Initial (VAR Schlange : SchlangenTyp);
(* Initialisierung der Schlange *)

PROCEDURE Loeschen (VAR Schlange : SchlangenTyp);
(* Loeschen der Schlange *)

END ADTSchlange.

```

Implementierung

```

IMPLEMENTATION MODULE ADTSchlange;                            (* Programm P3.5b *)
(* Demonstration eines ADTs:
   FIFO-Speicher fuer CARDINAL-Zahlen *)
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
TYPE SchlangenTyp = POINTER TO ElementType;
   ElementType    = RECORD
       Wert       : CARDINAL;
       Nachfolger : SchlangenTyp;
   END (* RECORD *);

PROCEDURE Bringen (VAR Schlange : SchlangenTyp;
                  Eintrag  : CARDINAL);
VAR NeuEintrag : SchlangenTyp;
BEGIN (* Bringen *) (* Vorn neues Glied in Kette einfuegen *)
  ALLOCATE (NeuEintrag, SIZE (ElementType));
  WITH NeuEintrag^ DO
    Wert := Eintrag;

```

```

    Nachfolger := Schlange;
  END (* WITH *);
  Schlange := NeuEintrag;
END Bringen;

PROCEDURE Holen (VAR Schlange : SchlangenTyp;
                 VAR Eintrag : CARDINAL);
VAR Position, VorPosition : SchlangenTyp;
BEGIN (* Holen *)      (* Letztes Glied der Kette entfernen *)
  IF Schlange <> NIL THEN
    Position := Schlange;
    VorPosition := NIL;
    WHILE Position^.Nachfolger # NIL DO
      VorPosition := Position;
      Position := Position^.Nachfolger;
    END (* WHILE *);
    Eintrag := Position^.Wert;
    DEALLOCATE (Position, SIZE (ElementTyp));
    IF VorPosition = NIL THEN
      Schlange := NIL;
    ELSE
      VorPosition^.Nachfolger := NIL;
    END (* IF *);
  END (* IF *);
END Holen;

PROCEDURE istLeer (Schlange : SchlangenTyp): BOOLEAN;
BEGIN (* istLeer *)
  RETURN (Schlange = NIL);
END istLeer;

PROCEDURE istVoll (Schlange : SchlangenTyp): BOOLEAN;
BEGIN (* istVoll *)
  RETURN FALSE;      (* Haldenueberlauf ist nicht abgefangen *)
END istVoll;

PROCEDURE Initial (VAR Schlange : SchlangenTyp);
BEGIN (* Initial *)
  Schlange := NIL;
END Initial;

PROCEDURE Loeschen (VAR Schlange : SchlangenTyp);
VAR Hilf : SchlangenTyp;
BEGIN (* Loeschen *)
  WHILE Schlange # NIL DO
    Hilf := Schlange;
    Schlange := Schlange^.Nachfolger;
    DEALLOCATE (Hilf, SIZE (ElementTyp));
  END (* WHILE *);
END Loeschen;

END ADTSchlange.

```

Test-Modul für ADT Schlange

```

MODULE ADTTest;                                     (* Programm P3.5c *)
(* Testrahmen fuer ADTSchlange: Deklaration von zwei Schlangen,
   die durch interaktive Bedienung gefüllt und geleert werden.
   Jede Eingabe besteht aus zwei Zahlen: a b
   a waehlt die Schlange 1 oder 2, 0 ist Programmabbruch.
   b (CARDINAL) wird in die Schlange gespeichert, wenn > 0;
   bei b = 0 wird der Schlange ein Element entnommen. *)
FROM ADTSchlange IMPORT      (* TYPE *) SchlangenTyp,
                             (* PROC *) Initial, Loeschen,
                             Bringen, Holen,
                             IstLeer, IstVoll;
FROM InOut      IMPORT      (* PROC *) ReadCard, WriteLn,
                             WriteCard, WriteString;
TYPE TestTyp = PROCEDURE (SchlangenTyp): BOOLEAN;
VAR Schlangenpaar : ARRAY [1 .. 2] OF SchlangenTyp;
    Nummer, Wert  : CARDINAL;

PROCEDURE WriteB (Value: BOOLEAN);
BEGIN (* Eine Prozedur zur Ausgabe boolscher Werte *)
    IF Value THEN WriteString ('T');
    ELSE WriteString ('F');    END;
END WriteB;

PROCEDURE Schritt (VAR eineSchlange : SchlangenTyp;
                   SollHolen       : BOOLEAN;
                   Zahl              : CARDINAL);

PROCEDURE TestSchritt (Test        : TestTyp;
                      stringteil : ARRAY OF CHAR);
(* glob. Var.: lesend eineSchlange, veraendert Zahl *)
BEGIN (* TestSchritt *)
    IF Test (eineSchlange) THEN
        WriteString ('Schlange ');    WriteCard (Nummer, 1);
        WriteString (' ist bereits '); WriteString (stringteil);
    ELSE
        IF SollHolen THEN
            Holen (eineSchlange, Zahl);
        ELSE
            Bringen (eineSchlange, Zahl);
        END (* IF *);
        WriteCard (Zahl, 4); WriteString (' ');
        WriteString (stringteil); WriteString (' = ');
        WriteB (Test (eineSchlange));
    END (* IF *);
    WriteLn;
END TestSchritt;

```



```

BEGIN (* Schritt *)
  IF SollHolen THEN
    TestSchritt (istLeer, 'leer');
  ELSE
    TestSchritt (istVoll, 'voll');
  END (* IF *);
END Schritt;

BEGIN (* Hauptprogramm *)
  WriteString ('*** Start Test ADTSchlange ***'); WriteLn;
  Initial (Schlangenpaar [1]); Initial (Schlangenpaar [2]);
  WriteString ('Leer (1) und Voll (2) : ');
  WriteB (istLeer (Schlangenpaar [1])); WriteString (' ');
  WriteB (istVoll (Schlangenpaar [2])); WriteLn; WriteLn;
  WriteString ('Gib jeweils Nummer (1 oder 2) ');
  WriteString ('und Zahl oder 0!');
  WriteLn;
  LOOP
    WriteString ('>'); ReadCard (Nummer); (* Prompten und
                                          Einlesen *)

    IF Nummer = 0 THEN EXIT END;
    IF Nummer <= 2 THEN
      WriteString (" #: "); ReadCard (Wert);
      Schritt (Schlangenpaar [Nummer], Wert = 0, Wert);
    END (* IF *);
    WriteLn;
  END (* LOOP *);
  Loeschen (Schlangenpaar [1]);
  Loeschen (Schlangenpaar [2]);
  WriteString ('*** Ende Test ADTSchlange ***'); WriteLn;
END ADTest.

```

Alternative Implementierung des ADTs Schlange

Bei gleichem Definitions-Modul sind auch ganz andere Implementierungen möglich, z.B. mit folgender Datenstruktur (vgl. Beispiel zur Kapselung):

```

IMPLEMENTATION MODULE ADTSchlange;          (* Programm P3.5d *)
(* FIFO-Speicher fuer CARDINAL-Zahlen:
   Version mit Ringpuffer *)
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
TYPE SchlangenTyp   = POINTER TO RingpufferTyp;
CONST MaxElemente   = 100;
      Feldlaenge     = MaxElemente+1; (* 1 Platz bleibt frei *)
TYPE SchlIndTyp     = [0 .. Feldlaenge-1];
      RingpufferTyp = RECORD
        Puffer : ARRAY SchlIndTyp OF CARDINAL;
        AnfangVoll,
        AnfangLeer : SchlIndTyp;
      END (* RECORD *);

usw.

```

ADT Stack

Ein *Stack (Keller)* ist eine variable Datenstruktur zur Verwaltung von mehreren Elementen des gleichen Typs nach dem LIFO-Prinzip.

Anschaulich kann man sich einen Stack als lineare Liste vorstellen, bei der alle Operationen Einfügen, Löschen und Abfrage jedoch nur an einem Ende der Liste ausgeführt werden. Auf Stacks sind i.a. folgende Operationen definiert:

Push (Element):	<i>Einfügen</i> eines neuen (dann obersten) Elementes,
Pop():	<i>Entfernen</i> des obersten Elementes,
Top():	Abfrage des <i>obersten</i> Elementes,
Empty():	Abfrage, ob der Stack <i>leer</i> ist.

In der folgenden Implementierung eines ADT Stack für CARDINAL-Zahlen werden außerdem noch Prozeduren zum Initialisieren und Löschen eines Stacks zur Verfügung gestellt.

Schnittstelle

```

DEFINITION MODULE ADTStack;                                (* Programm P3.6a *)
(* Demonstration eines ADTs:
   LIFO-Speicher fuer CARDINAL-Zahlen *)
TYPE StackTyp;                                             (* ein privater (opaker) Typ *)

PROCEDURE Push (VAR Stack : StackTyp;
                VAR Eintrag : CARDINAL);
(* Speichern des übergebenen Wertes als oberstes Stack-
   element *)

PROCEDURE Pop ( VAR Stack : StackTyp;
                VAR istLeer : BOOLEAN ) : CARDINAL;
(* Abfrage und Entfernen des obersten Stackelementes:
   wenn istLeer gleich FALSE, wird der Wert des obersten
   Stackelements zurueckgeliefert und das Element geloescht;
   sonst hat der Prozeduraufruf keine Wirkung und der Rueck-
   gabewert ist 0. *)

PROCEDURE Top ( VAR Stack : StackTyp;
                VAR istLeer : BOOLEAN ) : CARDINAL;
(* Abfrage des obersten Stackelementes:
   wenn istLeer gleich FALSE, wird der Wert des obersten
   Stackelementes zurueckgeliefert, sonst der Wert 0. *)

PROCEDURE Empty (Stack : StackTyp): BOOLEAN;
(* TRUE, genau wenn der Stack leer ist *)

```

```

PROCEDURE Initial (VAR Stack : StackTyp);
(* Initialisierung des Stacks *)

PROCEDURE Delete (VAR Stack : StackTyp);
(* Loeschen des Stacks, d.h. Loeschen aller Elemente *)

END ADTStack.

```

Implementierung

```

IMPLEMENTATION MODULE ADTStack;                                (* Programm P3.6b *)
(* Demonstration eines ADTs:
   LIFO-Speicher fuer CARDINAL-Zahlen *)
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
TYPE StackTyp      = POINTER TO ElementTyp;
   ElementTyp      = RECORD
       Wert          : CARDINAL;
       Nachfolger    : SchlangenTyp;
   END (* RECORD *);

PROCEDURE Push (VAR Stack      : StackTyp;
                Eintrag      : CARDINAL);
VAR NeuElement : StackTyp;
BEGIN (* Push *) (* Neuen Eintrag als 1. Element einfuegen *)
    ALLOCATE (NeuElement, SIZE (ElementTyp));
    WITH NeuElement^ DO
        Wert := Eintrag;
        Nachfolger := Stack;
    END (* WITH *);
    Stack := NeuElement;
END Push;

PROCEDURE Pop (VAR Stack      : StackTyp;
               istLeer      : BOOLEAN ) : CARDINAL;
VAR Eintrag : CARDINAL;
    Hilf    : StackTyp;
BEGIN (* Pop *) (* Erstes Element lesen und loeschen *)
    istLeer := Empty(Stack);
    IF istLeer THEN
        Eintrag := 0;
    ELSE
        Eintrag := Stack^.Wert;
        Hilf := Stack;
        Stack := Stack^.Nachfolger;
        DEALLOCATE (Hilf, SIZE (ElementTyp));
    END;
    RETURN Eintrag;
END Pop;

PROCEDURE Top (VAR Stack      : StackTyp;
               istLeer      : BOOLEAN ) : CARDINAL;
VAR Eintrag : CARDINAL;

```



```

BEGIN (* Top *)  (* Erstes Element lesen *)
  istLeer := Empty(Stack);
  IF istLeer THEN
    Eintrag := 0;
  ELSE
    Eintrag := Stack^.Wert;
  END;
  RETURN Eintrag;
END Top;

PROCEDURE istLeer (Stack : StackTyp): BOOLEAN;
BEGIN (* istLeer *)
  RETURN (Stack = NIL);
END istLeer;

PROCEDURE Initial (VAR Stack : StackTyp);
BEGIN (* Initial *)
  Stack := NIL;
END Initial;

PROCEDURE Delete (VAR Stack : StackTyp);
VAR Hilf : StackTyp;
BEGIN (* Delete *)
  WHILE Stack # NIL DO
    Hilf := Stack;
    Stack := Stack^.Nachfolger;
    DEALLOCATE (Hilf, SIZE (ElementTyp));
  END (* WHILE *);
END Delete;

END ADTStack.

```

Auf die Angabe des entsprechenden Test-Moduls für den ADT Stack wird hier verzichtet, da dieses die gleiche Struktur wie das Test-Modul für den ADT Schlange besitzt. Der wesentliche Unterschied besteht darin, daß für einen Stack eine Abfrage "IstVoll" wegen seiner potentiell unbeschränkten Kapazität (nur durch die Größe des Hauptspeichers begrenzt) entfallen kann.

3.2.4.3 Abstrakter Datentyp für große Zahlen

Ein weiteres Beispiel für einen ADT ist die Bearbeitung großer natürlicher Zahlen. Hier geht es darum, einige wenige, sehr große natürliche Zahlen zu berechnen, beispielsweise eine Fakultät, die den Rahmen der CARDINAL-Zahlen sprengt, wie z.B. Fakultät von 47 = 258 623 241 511 168 180 642 964 355 153 611 979 969 197 632 389 120 000 000 000.

Es kommt hier also nicht auf die Effizienz, sondern auf eine möglichst einfache Implementierung an.

Schnittstelle

```

DEFINITION MODULE Big;                                (* Programm P3.7a *)
(* Beispiel fuer ADT: Abstrakter Datentyp fuer grosse Zahlen
   BIG stellt einen ADT fuer grosse natuerliche Zahlen bereit
   Bei Ueberlauf wird als Ergebnis die groesstmoeegliche Zahl
   geliefert, ausserdem die Meldung "Ueberlauf" ausgegeben *)

TYPE BigNo;                                           (* Datentyp fuer grosse Zahlen, opak *)

PROCEDURE Create (VAR Zahl: BigNo);
(* Zahl existiert anschliessend, aber ist nicht definiert *)

PROCEDURE Enter (VAR Zahl: BigNo; Wert: CARDINAL);
(* besetzt existierende Zahl mit Wert, ist dann definiert *)

PROCEDURE Add (S1, S2: BigNo; VAR Summe: BigNo);
(* Addiert S1 und S2, liefert Summe
   S1 und S2 muessen definiert sein,
   Summe muss existieren, wird definiert. *)

PROCEDURE Mul (F1, F2: BigNo; VAR Produkt: BigNo);
(* Multipliziert F1 und F2, liefert Produkt
   F1 u. F2 muessen definiert sein,
   Produkt muss nicht existieren, wird definiert.*)

PROCEDURE Power (Base: BigNo; Exp: CARDINAL;
                VAR isPower: BigNo);
(* liefert in isPower Base hoch Exp
   Base muss def. sein, isPower muss exist., ist dann def. *)

PROCEDURE Out (Zahl : BigNo);
(* gibt definierte Zahl in einer Zeile des Bildschirms aus
   Die Zahl erscheint rechtsbuendig, o. fuehrende Nullen. *)

END Big.

```

Implementierung

Bei Betrachtung der Implementierung beachte man, daß dieses Programm nur zum *einmaligen* Gebrauch, nicht für eine Programmbibliothek geschrieben wurde. Natürlich könnte man die Operationen wesentlich effizienter realisieren.

```

IMPLEMENTATION MODULE Big;                            (* Programm P3.7b *)
FROM InOut  IMPORT (* PROC *) WriteLn, Write, WriteCard,
                  WriteString;
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
CONST MaxLng = 60;  (* 10 ** MaxLng muss ausreichen fuer
                    Typ CARDINAL *)

TYPE Digit  = [0 .. 9];
   StoreNo = ARRAY [1 .. MaxLng] OF Digit;
   BigNo   = POINTER TO StoreNo;

```

```

PROCEDURE Create (VAR Zahl : BigNo);
BEGIN (* Create *)
    ALLOCATE (Zahl, SIZE (StoreNo));
END Create;

PROCEDURE InitNo (VAR Zahl : BigNo);           (* intern *)
(* Zahl auf null setzen *)
VAR Index : CARDINAL;
BEGIN (* InitNo *)
    FOR Index := 1 TO MaxLng DO    Zahl^[Index] := 0;    END;
END InitNo;

PROCEDURE Put9 (VAR Zahl : BigNo);           (* intern *)
(* Zahl auf hoechste darstellbare Zahl setzen *)
VAR Index : CARDINAL;
BEGIN (* Put9 *)
    FOR Index := 1 TO MaxLng DO    Zahl^[Index] := 9;    END;
END Put9;

PROCEDURE CopyNo (No1: BigNo; VAR No2: BigNo); (* intern *)
(* Wert von No1 nach No2 kopieren *)
BEGIN (* CopyNo *)
    No2^ := No1^;
END CopyNo;

PROCEDURE Out (Zahl : BigNo);
CONST Blank = ' ';    (* zur besseren Lesbarkeit *)
VAR    Index          : CARDINAL;
        FuehrendeNull : BOOLEAN;
BEGIN (* Out *)
    FuehrendeNull := TRUE;    (* noch keine Ziffer ausgegeben *)
    FOR Index := 1 TO MaxLng DO
        (* Zahl in Dreiergruppen ausgeben. *)
        IF FuehrendeNull AND (Zahl^[Index] = 0)
            AND (Index < MaxLng) THEN
            Write (Blank);    (* fuehrende Nullen unterdruecken *)
        ELSE
            WriteCard (Zahl^ [Index], 1);    FuehrendeNull := FALSE;
        END (* IF *);
        IF ((MaxLng - Index) MOD 3) = 0 THEN Write (Blank); END;
        (* Dreiergruppen bilden *)
    END (* FOR *);
    WriteLn;
END Out;

PROCEDURE Times10 (Zahl1 : BigNo; VAR Zahl2 : BigNo);
(* Zahl2 erhaelt den zehnfachen Wert von Zahl1 *)
VAR Index : CARDINAL;
BEGIN (* Times10 *)
    IF Zahl1^[1] # 0 THEN
        WriteString ('*** Times10 Ueberlauf ***');    WriteLn;
        Put9 (Zahl2);
    END IF;
END Times10;
    
```



```

ELSE
  FOR Index := 2 TO MaxLng DO
    Zahl2^[Index-1] := Zahl1^[Index];
  END;
  Zahl2^[MaxLng] := 0;
END (* IF *);
END Times10;

PROCEDURE Enter (VAR Zahl : BigNo; Wert: CARDINAL);
VAR Index : CARDINAL;
BEGIN (* Enter *)
  InitNo (Zahl); (* damit ist die Zahl auf 0 initialisiert *)
  Index := MaxLng;
  WHILE Wert > 0 DO      (* Zahl konvertieren, beginnend mit
                        der niedrigsten Ziffer *)
    Zahl^[Index] := Wert MOD 10;
    Wert := Wert DIV 10;  DEC (Index);
  END (* WHILE *);
END Enter;

PROCEDURE Add (S1, S2: BigNo; VAR Summe : BigNo);
VAR Index,
    Zwischensumme (* Summe zweier Ziffern *) : CARDINAL;
BEGIN (* Add *)
  Zwischensumme := 0;
  Index := MaxLng;
  WHILE (Index > 0) DO  (* Ziffernweise von rechts nach
                        links addieren *)
    Zwischensumme := S1^[Index] + S2^[Index]
                  + Zwischensumme DIV 10 (* Uebertrag *);
    Summe^[Index] := Zwischensumme MOD 10;
    DEC (Index);
  END (* WHILE *);
  IF Zwischensumme > 9 THEN
    (* Ueberlauf in der hoechsten Ziffer *)
    Put9 (Summe);
    WriteString ('*** Add Ueberlauf ***');  WriteLn;
  END (* IF *);
END Add;

PROCEDURE Mul (F1, F2 : BigNo; VAR Produkt : BigNo);
VAR Index : CARDINAL;
    Prod : BigNo; (* Zwischenspeicher fuer Ergebnis *)
    F2No : StoreNo;
BEGIN (* Mul *)
  F2No := F2^;
  ALLOCATE (Prod, SIZE (StoreNo));  InitNo (Prod);
  Index := 1;                        (* 1. Ziffer ungleich 0 suchen *)
  WHILE (Index <= MaxLng) AND (F2No[Index] = 0) DO
    INC (Index);
  END;

```

```

    WHILE Index <= MaxLng DO
        Times10 (Prod, Prod);  (* Dezimal-Shift um eine Stelle *)
        WHILE F2No[Index] > 0 DO
            (* Multiplizieren durch wiederholtes Addieren. *)
            Add (F1, Prod, Prod);
            DEC (F2No[Index]);
        END (* WHILE *);
        INC (Index);
    END (* WHILE *);
    IF Produkt # NIL THEN
        DEALLOCATE (Produkt, SIZE (StoreNo));
    END (* IF *);
    Produkt := Prod;
END Mul;

PROCEDURE Power (    Base   : BigNo; Exp: CARDINAL;
                   VAR isPower: BigNo);
VAR BaseCopy : BigNo;
BEGIN (* Power *)
    ALLOCATE (BaseCopy, SIZE (StoreNo));
    CopyNo (Base, BaseCopy);  (* Entkopplung
                               Parameter/Resultat *)
    Enter (isPower, 1);
    WHILE Exp > 0 DO
        (* Potenzieren durch wiederholtes Multiplizieren. *)
        Mul (BaseCopy, isPower, isPower);
        DEC (Exp);
    END (* WHILE *);
    DEALLOCATE (BaseCopy, SIZE (StoreNo));
END Power;

END Big.

```

Die Verwendung der Variablen Prod und F2No in der Prozedur Mul sowie der Variablen BaseCopy in der Prozedur Power ist erforderlich, da bei der Implementierung eines ADTs als Zeiger auf eine Datenstruktur die Manipulation von Wert-Parametern auch Rückwirkung auf den aufrufenden Programmteil haben kann.

Grundsätzlich können die Prozeduren des ADTs auch als *Funktionsprozeduren* formuliert sein, um ein einzelnes Ergebnis zu liefern, denn es handelt sich ja stets um Zeiger. Das führt zu etwas eleganteren Formulierungen, beispielsweise für die Prozedur Add (die dann besser Summe heißt) zum Kopf

```

PROCEDURE Summe (S1, S2: BigNo): BigNo;
(* liefert Summe von S1 und S2 *)
(* S1 und S2 müssen definiert sein *)

```

Aus dem Prozeduraufruf in Mul Add (F1, Prod, Prod);
wird die Zuweisung Prod := Summe (F1, Prod);

Diese Lösung hat aber einen schwerwiegenden Nachteil: der Programmierer kann nun dem Laufzeitsystem keine Unterstützung bei der Speicherverwaltung mehr geben. Während der Aufruf von Add keine weitere Belegung der Halde bewirkte, führt die Wertzuweisung dazu, daß die bisher mit Prod bezeichnete Datenstruktur *implizit* freigegeben wird.

3.2.4.4 Abstrakter Datentyp Komplex

Komplexe Zahlen sind ein weiteres klassisches Beispiel für die Bildung von ADTen.

Schnittstelle

```

DEFINITION MODULE KomplexM;                                (* Programm P3.8a *)
(* KomplexM stellt Datenstruktur und Operat. fuer kompl. Zahlen
   Achtung, das Problem des Zahlenueber und -unterlaufs bei
   REAL-Operationen ist nicht abgefangen *)
TYPE Komplex;                                              (* Opaker Typ fuer komplexe Zahlen *)

PROCEDURE Erzeugen (VAR X: Komplex);
(* X entsteht als neue komplexe Zahl, exist. anschliessend *)

PROCEDURE Loeschen (VAR X: Komplex);
(* X wird als komplexe Zahl geloescht
   X muss existieren, existiert anschliessend nicht mehr *)

PROCEDURE Setzen (VAR X: Komplex; RT, IT: REAL);
(* X erhaelt RT als Realteil, IT als Imaginaerteil
   X muss existieren, ist anschliessend definiert *)

PROCEDURE RT (X: Komplex): REAL;
(* liefert den Realteil von X
   X muss definiert sein *)

PROCEDURE IT (X: Komplex): REAL;
(* liefert den Imaginaerteil von X
   X muss definiert sein *)

PROCEDURE Summe (X, Y: Komplex; VAR Z: Komplex);
(* Z erhaelt als Wert die Summe von X und Y
   X und Y muessen def. sein, Z muss exist., anschl. def. *)

PROCEDURE Differenz (X, Y: Komplex; VAR Z: Komplex);
(* Z erhaelt als Wert die Differenz von X und Y (d.h. X - Y)
   X und Y muessen def. sein, Z muss exist., anschl. def. *)

PROCEDURE Produkt (X, Y: Komplex; VAR Z: Komplex);
(* Z erhaelt als Wert das Produkt von X und Y
   X und Y muessen def. sein, Z muss exist., anschl. def. *)

```



```

PROCEDURE Quotient (X, Y: Komplex; VAR Z: Komplex);
(* Z erhaelt als Wert den Quotienten von X und Y (d.h.  $X / Y$ )
   Hat Y den Betrag null, so wird Z auf null gesetzt
   X und Y muessen def. sein, Z muss exist., anschl. def. *)

PROCEDURE Betrag (X: Komplex): REAL;
(* liefert den Betrag von X; X muss definiert sein *)

PROCEDURE Gleich (X, Y: Komplex): BOOLEAN;
(* TRUE, wenn X und Y exakt gleich
   X und Y muessen definiert sein *)

END KomplexM.

```

Implementierung

```

IMPLEMENTATION MODULE KomplexM; (* Programm P3.8b *)
FROM Storage IMPORT (* PROC *) ALLOCATE, DEALLOCATE;
FROM MathLib0 IMPORT (* PROC *) sqrt;
TYPE KomplexInt = RECORD Re, Im : REAL END;
    Komplex = POINTER TO KomplexInt;
(*-----*)
PROCEDURE Erzeugen (VAR X: Komplex);
BEGIN (* Erzeugen *)
    ALLOCATE (X, SIZE (KomplexInt));
END Erzeugen;

PROCEDURE Loeschen (VAR X: Komplex);
BEGIN (* Loeschen *)
    DEALLOCATE (X, SIZE (KomplexInt));
    X := NIL;
END Loeschen;
(*-----*)
PROCEDURE Setzen (VAR X: Komplex; RT, IT: REAL);
BEGIN (* Setzen *)
    X^.Re := RT; X^.Im := IT;
END Setzen;

PROCEDURE RT (X: Komplex): REAL;
BEGIN RETURN X^.Re; END RT;

PROCEDURE IT (X: Komplex): REAL;
BEGIN RETURN X^.Im; END IT;
(*-----*)
PROCEDURE Summe (X, Y: Komplex; VAR Z: Komplex);
BEGIN (* Summe *)
    Z^.Re := X^.Re+Y^.Re;
    Z^.Im := X^.Im+Y^.Im;
END Summe;

```

```

PROCEDURE Differenz (X, Y: Komplex; VAR Z: Komplex);
BEGIN (* Differenz *)
  Z^.Re := X^.Re-Y^.Re;
  Z^.Im := X^.Im-Y^.Im;
END Differenz;
(*-----*)
PROCEDURE Produkt (X, Y: Komplex; VAR Z: Komplex);
VAR ZInt: KomplexInt;      (* fuer den Fall X = Z oder Y = Z *)
BEGIN (* Produkt *)
  ZInt.Re := X^.Re * Y^.Re - X^.Im * Y^.Im;
  ZInt.Im := X^.Re * Y^.Im + X^.Im * Y^.Re;
  Z^ := ZInt;
END Produkt;

PROCEDURE Quotient (X, Y: Komplex; VAR Z: Komplex);
VAR ZInt : KomplexInt; (* fuer den Fall X = Z oder Y = Z *)
    Nenner : REAL;
BEGIN (* Quotient *)
  IF Betrag (Y) = 0.0 THEN          (* Division durch null *)
    Setzen (Z, 0.0, 0.0);
  ELSE
    WITH Y^ DO
      Nenner := Re * Re + Im * Im;
    END (* WITH *);
    ZInt.Re := (X^.Re * Y^.Re + X^.Im * Y^.Im) / Nenner;
    ZInt.Im := (X^.Im * Y^.Re - X^.Re * Y^.Im) / Nenner;
    Z^ := ZInt;
  END (* IF *);
END Quotient;
(*-----*)
PROCEDURE Betrag (X: Komplex): REAL;
BEGIN (* Betrag *)
  WITH X^ DO
    RETURN sqrt (Re*Re + Im*Im);
  END (* WITH *);
END Betrag;

PROCEDURE Gleich (X, Y: Komplex): BOOLEAN;
BEGIN (* Gleich *)
  RETURN (X^.Re = Y^.Re) AND (X^.Im = Y^.Im);
END Gleich;

END KomplexM.

```

Test-Modul für Komplex M

Der folgende Rahmen zeigt, daß mit Objekten eines ADTs vorsichtig umgegangen werden muß. Insbesondere sollten Wertzuweisung und Vergleich nicht verwendet werden, obwohl der Compiler dies nicht verhindert.

```
MODULE TestKomplex;                                (* Programm P3.8c *)
(* Anwendung des Moduls KomplexM, JL, 1986- 5-13
   Achtung, der Inhalt dieses Moduls ist voellig unsinnig. *)
FROM KomplexM IMPORT (* TYPE *) Komplex,
                    (* PROC *) Erzeugen, Setzen, RT, IT,
                        Summe, Differenz;
FROM InOut    IMPORT (* PROC *) WriteString, WriteLn;
VAR A, B, C : Komplex;

BEGIN (* TestKomplex *)
  Erzeugen (A);  Erzeugen (B);  Erzeugen (C);
  Setzen (A, 1.0, 0.0);  Setzen (B, 1.0, 0.0);
  IF A = B THEN                                (* unsinniger Vergleich *)
    C := A; WriteString ("Zweig 1"); (* und *)
  ELSE                                          (* unsinnige *)
    C := B; WriteString ("Zweig 2"); (* Wertzuweisungen *)
  END;
  WriteLn;
END TestKomplex.
```


4. Semantik, Verifikation und Test

Eines der zentralen Probleme der Informatik besteht darin sicherzustellen, daß ein Programm genau das bewirkt, was beabsichtigt war. Allgemein ist dieses Problem algorithmisch nicht lösbar; bis heute ist auch kein Verfahren bekannt, um Software realistischer Komplexität garantiert fehlerfrei zu konstruieren.

In dieser Situation müssen wir von Fall zu Fall versuchen, einen plausiblen Kompromiß zwischen dem Fehlerrisiko einerseits und dem alltäglichen Nutzen andererseits zu finden:

- Reduzieren wir mit hohem Aufwand und durch enge Randbedingungen die Zahl der Fehler, so wird die Software in vielen Fällen zu spät fertig, zu teuer und zu wenig komfortabel.
- Vernachlässigen wir dagegen die Korrektheit, weil wir vor allem viele Funktionen und höchste Leistung der Programme anstreben, so erhalten wir ein unzuverlässiges und nutzloses, unter Umständen gefährliches Resultat.

Die hohe Leistung moderner Rechner einerseits, die weitreichenden Wirkungen der Programme andererseits führen dazu, daß die relative Bedeutung der Korrektheit laufend steigt, nicht nur bei offenkundig sicherheitsrelevanten technischen Anwendungen wie Kraftwerksüberwachungen, sondern auch bei typischen Informationssystemen wie einem Auskunftssystem, das Fragen nach der Kreditwürdigkeit eines Kunden beantwortet. Natürlich müssen auch mittelbar beteiligte Programme, z.B. Übersetzer, korrekt sein.

Darum befassen wir uns in den Abschnitten 4.2 und 4.3 mit zwei Methoden, deren Anwendung - auf der Ebene des Programmcodes - zur Korrektheit beiträgt: mit der *Verifikation* und dem *Test*. Beide benötigen als Grundlage eine exakte Definition der beabsichtigten Funktion, d.h. die *Semantik* muß klar sein, was wir in Abschnitt 4.1 diskutieren.

Verifikation und Test sind komplementär: während die Verifikation darauf abzielt, die Korrektheit mit mathematischen Mitteln nachzuweisen, versucht man beim Test, durch Ausführung des Programms mit vorgegebenen Sollresultaten Fehler aufzudecken.

Beide Verfahren haben, wenn sie richtig verstanden sind, auch Auswirkungen auf die *Software-Konstruktion*. Es sollte aber von Beginn an klar sein, daß diese Maßnahmen nur im Zusammenhang eines umfassenden, alle Phasen der Entwicklung beeinflussenden Software Engineering möglich und sinnvoll sind.

4.1 Konzepte für die Semantikdefinition

4.1.1 Semantik: Begriff und Motivation

Die *Semantik* einer Sprache ist die Festlegung, welche Bedeutung die syntaktisch korrekten Sätze der Sprache haben (vgl. Kap. 1). Sie ergänzt also z.B. Grammatiken oder Syntaxdiagramme, durch die die Syntax definiert ist; ohne Semantik wissen wir nur, *ob* eine bestimmte Zeichenreihe ein Programm ist, aber nicht, welche Berechnung sie ggf. beschreibt.

Mangels formaler Beschreibungsmethoden wurde die Semantik von Programmiersprachen früher nahezu ausschließlich *informell* erklärt. Seit 1970 wurden die formalen Verfahren jedoch erheblich weiterentwickelt und damit auch praktisch anwendbar gemacht. Während aber eine (teilweise) formale Definition der Syntax, z.B. durch EBNF-Grammatiken, den meisten Informatikern geläufig, ja selbstverständlich ist, wird die entsprechende Behandlung der Semantik eher als typisch akademische, praxisferne Übung betrachtet. Gerade weil die gängigen Programmiersprachen vor allem an bekannte Notationen der Mathematik anknüpfen, meint man, die Wirkung auch ohne präzise Definition zu kennen. Dieses Problem verstärkt sich in dem Maße, in dem man Programmiersprachen in einer gewissen Fertigkeit beherrscht und sich beim Erlernen einer neuen Programmiersprache auf die Syntax konzentriert (in der Tat werden neue Sprachen oft "vergleichend" eingeführt, indem man nur die Unterschiede zu einer als bekannt vorausgesetzten Sprache anführt), ohne sich um die Semantik zu kümmern. Dieses "intuitive" Sprachverständnis versagt jedoch in Grenz- und Zweifelsfällen, wie sie beim Programmieren dann und wann entstehen; es bietet auch keine Grundlage zur Verifikation, also zur formalen Bearbeitung der Semantik.

Auch die Konstruktion von *Übersetzern* erfordert eine möglichst genaue Semantikdefinition. Ein Übersetzer wandelt Programme der einen Programmiersprache (*Quellprogramme*) in Programme einer anderen Programmiersprache (*Zielprogramme*) um. Quellprogramm und Zielprogramm müssen die gleiche Wirkung haben, wenn man sie auf einer Rechenanlage ausführt, d.h. sie müssen semantisch gleichwertig (*äquivalent*) sein.

Wir benötigen also eine exakt und formal definierte Semantik, um zweifelsfrei entscheiden zu können, wie ein Programm zu interpretieren ist, und um Übersetzer bauen zu können, die dieser Interpretation entsprechend das Programm in ein semantisch äquivalentes Zielprogramm umsetzen.

Beispiel

Gegeben sei der folgende Programmtext in einer PASCAL-Codierung.

```
program p (input, output);                (* Programm P4.1 *)
var n, i, s: integer;
begin
  read (n);    s:= 1;
  for i := 2 to n do s := s * i;
  write(s);
end.
```

Ohne Semantik ist dieser Text eine sinnleere Folge von Zeichen. Um seine Bedeutung zu klären, muß man wissen, um welche Programmiersprache es sich handelt und was darin die einzelnen Wörter und Zeichen besagen. Zum Beispiel hat “:=” die Bedeutung “Die Variable auf der linken Seite erhält den Wert des Ausdrucks auf der rechten Seite”. Ist die Bedeutung aller Zeichen bekannt, so kann man die Bedeutung des ganzen Programms bestimmen. Man erkennt dann, daß das Programm p - im Rahmen des INTEGER-Zahlenbereichs - die folgende Funktion berechnet (n! bezeichnet die Fakultät):

$$\begin{aligned} f_p(n) &= 1, \text{ falls } n \leq 0 \\ f_p(n) &= n!, \text{ falls } n > 0. \end{aligned}$$

Das heißt: Die Bedeutung von p ist die durch p berechnete Funktion, hier als f_p bezeichnet. Um den Schluß von p auf f_p nicht nur intuitiv, sondern nachvollziehbar und überprüfbar ziehen zu können, um also nicht nur zu vermuten, sondern auch zu beweisen, daß p die Funktion f_p berechnet, benötigt man Methoden der Verifikation, die auf der Definition der Semantik von Programmiersprachen aufbauen.

4.1.2 Grundprinzipien der Semantiknotationen

Zur Beschreibung der Semantik von Programmiersprachen gibt es verschiedene *Ansätze*. Die wichtigsten sind

- die Übersetzersemantik
- die operationale Semantik
- die denotationale Semantik
- die axiomatische Semantik.

Die Grundprinzipien dieser - nicht immer scharf unterscheidbaren - Ansätze sollen nachfolgend vorgestellt werden.

In den folgenden Beispielen ist die Beschränkung des Zahlenbereichs im Computer nicht berücksichtigt; in der Praxis ist das natürlich nicht sinnvoll.

Übersetzersemantik

Dieser Ansatz geht von der Idee aus, die Bedeutung von Anweisungen einer *neu* zu definierenden Programmiersprache P_N durch die Bedeutung von Anweisungen einer bereits bezüglich Syntax und Semantik bekannten "*alten*" Programmiersprache P_A festzulegen. Das Grundprinzip ist also die formale *Zurückführung* einer jeden Anweisung von P_N in eine äquivalente Anweisung oder Anweisungsfolge aus P_A (1:n-Abbildung, d.h. einer Anweisung aus P_N entspricht eine Folge von P_A -Anweisungen).

Die Bedeutung eines P_N -Programms wird dann bei der Übersetzersemantik als Bedeutung des durch Zurückführung gewonnenen, als äquivalent definierten P_A -Programms festgelegt. Dieser Nachweis muß durch formale Methoden (z.B. *Simulation* einer die Sprache P_N erkennenden Maschine durch eine die Sprache P_A erkennenden Maschine) geführt werden.

Eine praktische Verwendung der Übersetzersemantik findet man, wie der Name schon suggeriert, im *Compilerbau*.

Operationale Semantik

Man definiert bei der operationalen Semantik eine (mathematische) *Maschine* (auch *Automat* genannt) für eine Programmiersprache, die aus Eingabedaten durch schrittweise Abarbeitung des Programms die Ausgabedaten erzeugt. Das Adjektiv "operational" verdeutlicht, daß ein konkretes *Verfahren* angegeben wird, *wie* der Ausgabewert durch eine Folge von Rechenschritten (Operationen) effektiv aus der Eingabe erzeugt werden kann. Zur exakten Definition der Maschine eignen sich verschiedene Modelle.

Eine praktische Verwendung der operationalen Semantik findet man in der Entwicklung von Sprachinterpretern, z.B. für die Sprache BASIC. Wir werden den operationalen Semantikansatz in 4.1.3 noch ausführlich vorstellen.

Denotationale Semantik

Beim operationalen Semantikansatz scheint insofern ein unverhältnismäßig großer technischer Aufwand notwendig zu sein, als im Prinzip ja nicht alle Zwischenzustände bei der Abarbeitung eines Programms wichtig sind, sondern nur das *Endergebnis* der durch ein Programm berechneten *Funktion* interessiert.

Bei der denotationalen Semantik wird daher von einem konkreten Maschinenmodell (wie es bei der operationalen Semantik notwendig war) abstrahiert und nur die *Wirkung* von Anweisungen in Form veränderter *Variablenbelegungen* untersucht. Eine praktische Verwendung der denotationalen Semantik findet man ebenfalls in der Entwicklung von Sprachinterpretern, z.B. für die Sprache LISP.

Wenn Z_P also die Menge aller möglichen Zustände ist, die bei der Ausführung eines Programms P durchlaufen werden können

(formal: $Z_P \subseteq W(v_1) \times W(v_2) \times \dots \times W(v_n)$, wobei $v_1, \dots, v_n$ die in P auftretenden Variablen und $W(v_i)$ der Wertebereich von v_i , d.h. die Menge aller Werte, die v_i annehmen kann, sind),

dann ist die Wirkung einer Anweisung $a \in P$ eine Abbildung $F[a] : Z_P \rightarrow Z_P$, die einem Zustand $z \in Z_P$ einen Folgezustand $z' \in Z_P$ zuordnet. z' unterscheidet sich von z nur durch die von a veränderten Variablenbelegungen.

Beispiel: Sei $V = \{x, y\}$ und $P = \dots; x := y + 1; \dots$.

Falls vor Erreichen der Zuweisung der Zustand $z = (x = \text{undef.}, y = 5)$ gilt, gilt nach der Anweisung der Zustand $z' = (x = 6, y = 5)$.

Formal: $F[x := y + 1](\text{undef.}, 5) = (6, 5)$.

F ordnet also einer Anweisung (hier: $x := y + 1$) eine *Zustandsänderung* als Bedeutung in Form veränderter Variablenbelegung zu. F wird daher auch *semantische Funktion* genannt.

Das Grundprinzip der denotationalen Semantik besteht nun darin, für *alle* möglichen Anweisungen einer neu zu definierenden Programmiersprache deren jeweilige semantische Funktion festzulegen, ohne auf eine zugrundeliegende Sprache (wie bei der Übersetzersemantik) oder ein basierendes Maschinenmodell (wie bei der operationalen Semantik) angewiesen zu sein. Natürlich müssen die semantischen Funktionen insgesamt "verträglich" (d.h. vollständig und widerspruchsfrei) definiert sein.

Axiomatische Semantik

In Fortführung des Abstraktionsprozesses beim Übergang von der operationalen zur denotationalen Semantik wird bei der axiomatischen Semantik nur noch auf *Eigenschaften* von Zuständen und nicht mehr - wie bei der denotationalen - auf konkrete Zustände selbst Bezug genommen.

Statt einer semantischen Funktion, die Zustände auf Zustände abbildet, wird bei der axiomatischen Semantik die Wirkung einer Anweisung a durch ein Tripel $(\{P\}, a, \{Q\})$ definiert, wobei P und Q *Prädikate* (Bedingungen) sind. Wenn P vor Ausführung von a gilt (P heißt deshalb auch *Vorbedingung*), dann muß Q nach Ausführung von a gelten (Q heißt deshalb auch *Nachbedingung*), allerdings unter der wichtigen Voraussetzung, daß a terminiert, d.h. nach endlich vielen Schritten hält.

Beispiele: 1. $-1000 < x \leq 0 \{ x := x - 1 \} -1000 \leq x < 0$,

2. $2 \leq y < 999 \{ \text{if } x \neq 0 \text{ then } y := y + 1 \text{ else } y := y - 1 \text{ end} \}$
 $(3 \leq y \leq 999 \wedge x \neq 0) \vee (1 \leq y \leq 997 \wedge x = 0)$.

Eine praktische Verwendung der axiomatischen Semantik findet man u.a. in sogenannten "Wissensbasierten Systemen", einem Grundlagengebiet der *Künstlichen Intelligenz* (KI).

Der axiomatische Semantikansatz ist geeignet, Eigenschaften von Programmen zu beweisen, z.B. deren *Korrektheit* bezüglich einer vorgegebenen, spezifizierten Funktion. Diese *Verifikationstechnik* werden wir im Abschnitt 4.2 "Spezifikation und Verifikation von Programmen" detailliert kennenlernen.

Zusammenfassung der Semantikansätze

Semantikansatz	Grundprinzip: Bedeutung einer Anweisung a einer neu zu definierenden Programmiersprache P_N wird durch ...	Abstrahierungsvorgang
Axiomatische Semantik	Tripel $(\{P\}, a, \{Q\})$ definiert: wenn P "vor" a gilt, dann gilt "nachher" Q .	Abstrahierung von konkreten Zuständen
Denotationale Semantik	semantische Funktion F mit $F[a] z = z'$ definiert, wobei z' sich von z durch die "durch" a veränderten Variablen unterscheidet.	Abstrahierung vom konkreten Maschinenmodell
Operationale Semantik	math. Maschine M definiert: M ändert ihre Konfiguration bei effektiver Ausführung von a .	Verzicht auf Existenz einer bekannten Sprache, aber Abhängigkeit von einem Maschinenmodell
Übersetzersemantik	Zurückführung von a auf Anweisung(en) einer bereits bekannten Sprache definiert.	keine Abstrahierung, Nachteil: Abhängigkeit von (der Semantik) einer bekannten Sprache

4.1.3 Ein Beispiel für die operationale Semantik

Die Programmiersprache MINI

Der Ansatz der operationalen Semantik soll anhand einer einfachen Programmiersprache (MINI genannt) etwas ausführlicher erläutert werden. In MINI gibt es nur Variablen des Typs INTEGER, die mit den verfügbaren Anweisungen eingelesen, verändert und ausgegeben werden können.

Die *Syntax* von MINI sei durch die folgenden Produktionsregeln einer BNF-Grammatik gegeben (ϵ bezeichnet die leere Anweisung, ein senkrechter Strich ("|") alternative Ableitungsmöglichkeiten auf der rechten Seite einer Regel).

Programm $\rightarrow$ "read" Variablenliste ";" Anweisung ";" "write" Variablenliste ".".
 Variablenliste $\rightarrow$ Bezeichner | Bezeichner, Variablenliste.
 Bezeichner $\rightarrow$ "a" | "b" | "c" | ... | "z".
 Anweisung $\rightarrow$ ϵ
 | Bezeichner ":" "=" "0"
 | Bezeichner ":" "=" Bezeichner "+1"
 | Bezeichner ":" "=" Bezeichner "-1"
 | Anweisung ";" Anweisung
 | "if" Bezeichner "=" "0" "then" Anweisung "else" Anweisung
 "end"
 | "while" Bezeichner "=" "0" "do" Anweisung "end".

Kommentare, Zeilenwechsel usw. sind hier nicht formal definiert, sollen aber ebenfalls (ohne semantische Wirkung) in der Sprache enthalten sein.

Beispiel: Das folgende Programm in MINI liefert die Summe zweier Zahlen a und b, falls $b \geq 0$ ist; andernfalls terminiert es nicht.

```

read x, y;                                     (* Programm P4.2 *)
while y  $\neq$  0 do x := x + 1; y := y - 1 end;
write x.

```

Definition einer mathematischen Maschine

Eine *mathematische Maschine* M ist ein 7-Tupel $M = (I, O, K, \alpha, \omega, \tau, \pi)$; darin sind

I	die Eingabe, d.h. <i>Programmanweisungen</i> und <i>Eingabedaten</i> ,
O	eine Menge von <i>Ausgabedaten</i> ,
K	eine Menge von <i>Konfigurationen</i> ,
$\alpha: I \rightarrow K$	eine <i>Eingabefunktion</i> ,
$\omega: K \rightarrow O$	eine <i>Ausgabefunktion</i> ,
$\tau: K \rightarrow K$	eine <i>Übergangsfunktion</i> ,
$\pi: K \rightarrow \{0, 1\}$	ein <i>Halteprädikat</i> .

Nachfolgend wird eine Maschine konstruiert, die Programme der Programmiersprache MINI ausführt (interpretiert).

Die Menge der Konfigurationen K dieser Maschine definiert man formal als kartesisches Produkt $K = A \times Z$, wobei gilt:

- A ist die Menge aller *Anweisungsfolgen* von MINI,
 Z ist die Menge der *Zustände*, die ein Programm annehmen kann.

Eine Konfiguration ist also durch die zu bearbeitende Anweisungsfolge und den aktuellen Maschinenzustand charakterisiert.

Der *Zustand* einer Programmausführung ist durch die Menge der Bezeichner der Programmvariablen und deren Werte bestimmt. Ein Zustand $z \in Z$ läßt sich also durch eine *Tabelle* beschreiben, die zu jedem Bezeichner seinen aktuellen Wert enthält. Eine Endkonfiguration liegt vor, wenn alle Anweisungen abgearbeitet sind, d.h. wenn nur noch die leere Anweisung ϵ folgt. Daher gilt: $\pi(a, z) = 1$ genau dann, wenn $a = \epsilon$ ist.

Um A und Z nicht weiter formalisieren zu müssen, wird vorausgesetzt, daß

- bei jeder Anweisung $a \in A$ die Anweisung bis zum nächsten Semikolon (sofern vorhanden) erkannt werden kann;
- es in Z einen Anfangszustand z_0 gibt;
- jedem Zustand $z \in Z$ und jedem Bezeichner $b \in B$ einer Variablen ein Wert $z(b)$ zugeordnet ist, der den Wert der Variablen b im Zustand z angibt; für alle $b \in B$ gilt $z_0(b) = 0$, d.h. zu Beginn eines Programms haben alle Variablen den Wert 0;
- jedem Zustand $z \in Z$, allen Bezeichnern $b_1, b_2 \in B$ und jeder ganzen Zahl $w \in \mathbb{Z}$ ein neuer Zustand $z \langle b_1 \leftarrow w \rangle$ zugeordnet wird, wobei gilt:
 $z \langle b_1 \leftarrow w \rangle (b_2) = w$, falls $b_1 = b_2$, und $z(b_2)$ sonst;
 $z \langle b_1 \leftarrow w \rangle$ bezeichnet also den Zustand, den man erhält, wenn man im Zustand z den Wert der Variablen b_1 auf w setzt; die Werte aller übrigen Variablen bleiben unverändert.

Eine operationale Semantik für MINI

Die für die Definition der operationalen Semantik zentrale Übergangsfunktion τ kann man wie folgt beschreiben: $\tau: (a, z) \rightarrow (a', z')$.

Die einzelnen Werte von τ gibt die nächste Tabelle an; a, a_1 und a_2 stehen dabei für beliebige Anweisungen, b_1, b_2 usw. für beliebige Bezeichner. Dabei genügt es, immer nur *zwei* aufeinanderfolgende Anweisungen für a zu betrachten, da sich durch iterative Fortsetzung damit *beliebige* Anweisungsfolgen konstruieren lassen.

Die Tabelle gibt genau die schrittweise Abarbeitung von Anweisungen wieder: Wenn z.B. die Anweisung " $b_1 := b_2 + 1; a$ " (3. Zeile) auszuführen ist, dann wird die zu b_1 gehörende Komponente eines Zustands z auf den Wert von $b_2 + 1$ gesetzt. Danach verbleibt nur noch die Anweisung a (letzte Spalte der Tabelle) zur weiteren Interpretation.

alte Anweisungsfolge	alter Zustand	neuer Zustand	verbleibende Anweisungsfolge
$\epsilon; a$	z	z	a
$b := 0; a$	z	$z \langle b \leftarrow 0 \rangle$	a
$b_1 := b_2 + 1; a$	z	$z \langle b_1 \leftarrow z(b_2) + 1 \rangle$	a
$b_1 := b_2 - 1; a$	z	$z \langle b_1 \leftarrow z(b_2) - 1 \rangle$	a
if $b = 0$ then a_1 else a_2 end; a	z	z	$a_1; a,$ falls $z(b) = 0$ $a_2; a,$ falls $z(b) \neq 0$
while $b \neq 0$ do a_1 end; a_2	z	z	$a_2,$ falls $z(b) = 0$ $a_1;$ while $b \neq 0$ do a_1 end; $a_2,$ falls $z(b) \neq 0$

Die letzte Zeile der Tabelle ist folgendermaßen zu deuten: Falls die while-Bedingung nicht erfüllt ist ($b = 0$), wird nur die leere Anweisung (ϵ) ausgeführt, und man macht unmittelbar mit a_2 weiter. Im anderen Fall führt man einmal den Rumpf a_1 der while-Schleife aus und anschließend erneut die gesamte Schleife.

Die *Eingabe* des Interpreters besteht aus dem Programm und den Daten.

MINI-Programme haben die Form "read $b_1, \dots, b_r; a_1; a_2; \dots a_n$; write $c_1, \dots, c_s$." Eingabedaten bzw. Ausgabedaten sind Folgen der Form $b_1, \dots, b_r$ bzw. $c_1, \dots, c_s$. Bezeichnet P die Menge aller korrekten Programme, so ist $I = P \times \mathbb{Z}^*$ und $O = \mathbb{Z}^*$. $\mathbb{Z}^*$ ist die Menge aller endlichen Folgen ganzer Zahlen.

Die Eingabe I des Interpreters besteht also aus allen Paaren $(p, x_1 \dots x_r)$, wobei p ein Programm in der Programmiersprache MINI ($p \in P$) und $x_1 \dots x_r \in \mathbb{Z}^*$ die Eingabedaten bezeichnet. Die Ausgabemenge O besteht aus der Menge aller endlichen Folgen ganzer Zahlen, den Ausgabewerten eines Programms.

Unter Verwendung dieser Bezeichnungsweise sind die Eingabefunktion $\alpha: I \rightarrow K$ und die Ausgabefunktion $\omega: K \rightarrow O$ wie folgt definiert:

$$\alpha(p, x_1, \dots, x_r) = (a; \epsilon, z_0 \langle b_1 \leftarrow x_1 \rangle \dots \langle b_r \leftarrow x_r \rangle),$$

$$\omega(\epsilon, z) = (z(c_1), \dots, z(c_s)).$$

Dabei sind $c_1, \dots, c_s$ die im Zustand z kodierten Bezeichner der Programmvariablen, die in p hinter write aufgelistet sind.

Man beachte, daß a die in p enthaltene Folge von Anweisungen (ohne read- und write-Anweisungen) ist. Die obere Gleichung besagt im wesentlichen, daß im Zustand z_0 (alle Variablen gleich null) die Variablen $b_1, \dots, b_r$ mit den Eingabewerten $x_1, \dots, x_r$ belegt werden (*Initialisierung*) und daß a um die leere Anweisung ϵ ergänzt wird. Die Funktion ω gibt die Werte der Variablen $c_1, \dots, c_s$ im Zustand z aus.

Die Berechnung kann also nachvollzogen werden, indem man zunächst α anwendet, dann τ , bis nur noch ϵ übrigbleibt, d.h. bis $\pi(a, z) = 1$ gilt, und schließlich ω .

Der Interpreter ist damit *vollständig* definiert.

Die *berechnete Funktion* $f : P \times \mathbb{Z}^* \rightarrow \mathbb{Z}^*$ ist die Semantik von MINI.

Beispiel: Man betrachte das folgende Programm p (mit der Eingabe 2, 1):

```

read x, y;                                (* Programm P4.3 *)
if y = 0 then x := 0
      else while y ≠ 0 do y := y - 1 end
end;
write x, y.

```

Durch α wird folgende Startkonfiguration hergestellt:

(if y = 0 then x := 0 else
 while y ≠ 0 do y := y - 1 end end; ϵ, z_1). Zustand $z_1(x) = 2, z_1(y) = 1$.

Da $z_1(y) \neq 0$ ist, liefert die Übergangsfunktion τ in einem Schritt die Folgekonfiguration:

(while y ≠ 0 do y := y - 1 end; ϵ, z_1). Keine Zustandsänderung.

Als nächste Konfiguration erhält man durch einmaliges Ersetzen der while-Anweisung:

(y := y - 1; while y ≠ 0 do y := y - 1 end; ϵ, z_1). Keine Zustandsänderung.

Nun wird durch die Subtraktion zum ersten Mal auch der Zustand verändert:

(while y ≠ 0 do y := y - 1 end; ϵ, z_2). Zustand $z_2(x) = 2, z_2(y) = 0$.

Wegen $y = 0$ folgen die Konfigurationen ($\epsilon; \epsilon, z_2$) und (ϵ, z_2).

Es ist eine *Endkonfiguration* erreicht. Das Ergebnis lautet daher:

$\omega(\epsilon, z_2) = (2, 0)$, d.h. es werden die Zahlen 2 und 0 ausgegeben.

Durch algebraische Anwendung der Transformationen kann man zeigen, daß das Programm p folgende Funktion berechnet:

$$f_p: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \times \mathbb{Z} \text{ mit}$$
$$f_p(x, y) = \begin{cases} (x, 0), & \text{falls } y > 0 \\ (0, 0), & \text{falls } y = 0 \\ \text{undefiniert,} & \text{falls } y < 0. \end{cases}$$

4.2 Spezifikation und Verifikation von Programmen

In diesem Abschnitt soll gezeigt werden, daß es formale Möglichkeiten gibt, Programme zu *spezifizieren* (“Was soll das Programm leisten?”) und zu *verifizieren* (“Tut das Programm das, was es leisten soll?”).

Es sollen hier einige *Prinzipien* vorgestellt werden, die in der Programmierung - auch ohne formale Verifikation - hohe Bedeutung haben. Die wesentlichen Punkte sind:

- Zwischen der Programmiersprachen-Definition, der Programm-Spezifikation und der Programm-Verifikation besteht ein enger *Zusammenhang*.
- Korrektheit kann in ein Programm konstruktiv “*eingebaut*” werden.
- Trotz der Dynamik, mit der Algorithmen ausgeführt werden, kann ihre Wirkung *statisch* beschrieben werden.

Vorbemerkungen: Als Beispiel einer Programmiersprache diene MODULA-2. Generell gilt für diesen Abschnitt, daß das Problem des *Zahlenüberlaufs* ignoriert wird, wo es nicht speziell erwähnt ist. Wir tun also so, als sei der INTEGER-Bereich unbeschränkt.

4.2.1 Vor- und Nachbedingungen

Von jeder Berechnung, beispielsweise in einer Funktionsprozedur, erwarten wir ein bestimmtes Ergebnis; dieses wird als *Zusicherung* (*postcondition*) formuliert. Für die Funktionsvorschrift *Doppelt* (aus Abschnitt 2.2.4) lautet sie z.B.: $\text{Doppelt}(x) = 2 \cdot x$.

Man beachte, daß die Übereinstimmung mit der Realisierung keineswegs zwingend ist; das Ergebnis könnte ebenso gut durch Addition oder auf andere Weise berechnet werden. Hier geht es *nur* um eine Aussage über das *Resultat*.

Damit die Zusicherung auch gilt, müssen bestimmte *Vorbedingungen* (*preconditions*) erfüllt sein, z.B. $x \in \text{INTEGER} \wedge |x| \leq \text{MAX}(\text{INTEGER}) / 2$.

Man beachte, daß es sich hier um eine *Aussage*, nicht um ein Programm handelt. Darum sind MODULA-2-fremde Zeichen und die Division mit “/” hier zulässig. Auch *strengere* Vorbedingungen stellen ein korrektes Ergebnis sicher, z.B. $x \in \text{INTEGER} \wedge 0 \leq x \leq \text{MAX}(\text{INTEGER}) / 4$.

Um eine Funktionsvorschrift voll “auszuschöpfen”, suchen wir aber die *schwächste Vorbedingung* (*weakest precondition* oder *wp*).

Eine Funktionsvorschrift läßt sich bezüglich ihres *Effektes* also beschreiben durch

- die Zusicherung für das Ergebnis (Q) und
- die Vorbedingung (P).

Diese Angaben gehören in den *Kopfkommentar* eines entsprechenden Algorithmus, und sie spielen bei der Verifikation eine zentrale Rolle.

Beispiele ("≡" bezeichnet die logische Äquivalenz zwischen zwei Prädikaten):

1. $\text{Doppelt } Q \equiv \text{Doppelt } (x) = 2 \cdot x$
 $P \equiv x \in \text{INTEGER} \wedge |x| \leq \text{MAX}(\text{INTEGER}) / 2$
 Kommentar: Liefert das Doppelte einer Zahl, deren Betrag MAX (INTEGER) / 2 nicht übersteigt.

2. $\text{SQRT } Q \equiv \text{SQRT } (x) \approx \sqrt{x}$
 $P \equiv x \in \text{REAL} \wedge x \geq 0$
 Kommentar: Liefert die Quadratwurzel für nichtnegative Parameter.
 Die Zusicherung muß, wenn sie exakt formuliert werden soll, auf die Ungenauigkeit der REAL-Arithmetik Rücksicht nehmen:

$$Q \equiv |\text{SQRT } (x) - \sqrt{x}| < \sqrt{x} / 1000$$

Dadurch wird ein maximaler relativer Fehler von 1‰ beschrieben.

Notation, Definitionen

Vor- und Nachbedingungen sollen nun formal eingeführt werden.

Sei S ein Programm oder der Teil eines Programms. Sei ferner P eine Aussage über die Variablen des Programms, die vor Ausführung von S gilt, Q eine Aussage, die nach Ausführung von S gilt (Parameter werden im weiteren nicht von Variablen unterschieden, P und Q können also auch Parameter betreffen). Dann beschreiben wir diesen Zusammenhang durch

$$P \{ S \} Q$$

P wird als *Vorbedingung*, Q als *Nachbedingung* bezeichnet.

Wir können diese Aussage (nachfolgend A genannt) je nach Kontext für drei verschiedene Zwecke verwenden:

- (a) Ist S eine einzelne Anweisung, so dient A der *Definition* (Abk. DEF) dieser Anweisung. Indem wir solche Definitionen für alle Anweisungen einer Programmiersprache geben, definieren wir ihre *axiomatische Semantik*. Für

denjenigen, der einen Übersetzer baut, ist A die Vorgabe (*Übersetzer-Spezifikation*).

- (b) Ist S ein Programm (oder ein Programm-Fragment), so kann A aus den Definitionen der einzelnen Befehle *synthetisiert* werden. A ist dann die *Beschreibung* (Abk. BES) des Programms.
- (c) Ist S ein noch zu schaffendes Programm (oder Programm-Fragment), so ist A die *Spezifikation* (Abk. SPEC) des Programms.

Im allgemeinen klärt der Kontext, welche Bedeutung gemeint ist. Wo Unklarheiten entstehen könnten, zeigen wir durch den Index DEF, BES oder SPEC an, was gemeint ist.

Den Nachweis, daß Spezifikation und Beschreibung verträglich sind, bezeichnet man als *Verifikation* des Programms (*Programm-Beweis*). Formal: ein Programm ist verifiziert, wenn wir für dieses Programm beweisen können, daß

$$PSPEC \Rightarrow PBES \wedge QBES \Rightarrow QSPEC.$$

Durch die spezifizierte Vorbedingung muß also die Vorbedingung des realisierten Programms impliziert sein, und die Nachbedingung des realisierten Programms muß die spezifizierte Nachbedingung implizieren. Da wir natürlich daran interessiert sind, mit einem Programm eine bestimmte Wirkung zu erzielen, ist uns in der Regel QSPEC *vorgegeben*; in vielen Fällen wird unsere Aufgabe einfacher, wenn wir QSPEC zu QBES verallgemeinern (d.h. QBES ist eine einfache, allgemeine Aussage, in der QSPEC enthalten ist). Suchen wir nach der Vorbedingung PBES, so wünschen wir, diese so schwach wie möglich zu machen, wir suchen darum nach der *schwächsten Vorbedingung* eines Programms S zur Erlangung von QBES und schreiben

$$PBES \equiv wp(S, QBES).$$

Wir gehen also bei der Ableitung einer Programmbeschreibung nach Möglichkeit *rückwärts* vor, von den angestrebten Nachbedingungen zu den unvermeidlichen Vorbedingungen ("wir bemühen uns, etwas, das wir uns wünschen, zu einem möglichst niedrigen Preis zu erhalten").

Ein Programm ist um so *flexibler* und robuster, je weniger Vorbedingungen es stellt; ideal ist die Vorbedingung TRUE, die nicht durch irgendwelche speziellen Daten gefährdet werden kann. Das andere Extrem ist die schwächste Vorbedingung FALSE: In diesem Fall liefert das Programm unter keinen Umständen das richtige Ergebnis.

Später wird sich zeigen, daß in bestimmten Situationen der umgekehrte Ansatz Vorteile hat, also der Weg von einer Vorbedingung zu der *stärksten Nachbedingung* ("wir bemühen uns, für einen bestimmten Einsatz möglichst viel zu bekommen").

Beispiel (in natürlicher Sprache):

In einem Feld F mit der Indizierung 1 bis 1.000 und dem Elementtyp **INTEGER** kommen nur Werte im Bereich 0 bis 20 vor, jeder davon mehrfach. Alle Elemente eines bestimmten Wertes $a \in \{0, \dots, 20\}$ sollen im Feld gesucht werden.

Aus der Problemstellung folgt die Nachbedingung

$$Q_{SPEC} \equiv i \in \{1, \dots, 1.000\} \wedge F[i] = a.$$

Zur Realisierung ist es einfacher, das *erste* Vorkommen (oder das *letzte*) zu suchen:

$$Q_{BES} \equiv Q_{SPEC} \wedge (F[j] = a \Rightarrow j > i).$$

Q_{BES} impliziert also Q_{SPEC} .

Vorbedingung ist offenbar, daß a in F vorkommt:

$$P_{BES} \equiv \exists i \in \{1, \dots, 1.000\}: F[i] = a \wedge a \in \text{INTEGER}.$$

Dies ist durch die Problemstellung impliziert:

$$P_{SPEC} \equiv \exists i, j \in \{1, \dots, 1.000\}: F[i] = F[j] = a \wedge i \neq j \wedge a \in \{0, \dots, 20\}.$$

Also gilt: $P_{SPEC} \Rightarrow P_{BES}$.

4.2.2 Semantikdefinition einiger Anweisungen

Nachfolgend werden einige Anweisungen jeweils durch ihre schwächste Vorbedingung definiert. Diese Definitionen können später als *Axiome* verwendet werden.

Die leere Anweisung

Die leere Anweisung wird hier zur besseren Lesbarkeit durch ϵ dargestellt. In Programmen kommt sie immer dann vor, wenn ein redundantes Semikolon steht.

Semantik: $wp(\epsilon, Q) \equiv Q$

Informell: Was nach Ausführung von ϵ gelten soll, muß vorher bereits gelten.

Die Wertzuweisung

Die Semantik der Wertzuweisung ist kompliziert, wenn man alle *Fehlersituationen* berücksichtigt. Hier wird vorausgesetzt, daß der Ausdruck auf der rechten Seite einen definierten, in endlich vielen Schritten berechenbaren Wert hat und daß dieser mit dem Typ der Variablen auf der linken Seite kompatibel ist.

Semantik: $wp(x := E, Q(x)) \equiv \exists E \wedge E \in \text{TYP}(x) \wedge Q(E)$

(meist reduziert auf: $wp(x := E, Q(x)) \equiv Q(E)$)

Informell: E muß definiert und typkompatibel mit x sein. Dann ist die Vorbedingung gleich der Nachbedingung, in der alle Vorkommen von x durch den Ausdruck E ersetzt sind. E kann x enthalten (keine rekursive Ersetzung).

Man beachte, daß das Problem der REAL-Arithmetik hier nicht berücksichtigt wurde.

Die Sequenz

Semantik: $wp(S_1; S_2, Q) \equiv wp(S_1, wp(S_2, Q))$

Informell: Die schwächste Vorbedingung der letzten Anweisung ist die Nachbedingung der vorletzten (d.h. hier der ersten) Anweisung. Die Semantik einer Sequenz beliebig vieler Anweisungen ergibt sich durch rekursive Schachtelung.

Die Alternative

Semantik: $wp(\text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END}, Q)$

$$\equiv (B \Rightarrow wp(S_1, Q)) \wedge (\neg B \Rightarrow wp(S_2, Q))$$

$$\equiv (B \wedge wp(S_1, Q)) \vee (\neg B \wedge wp(S_2, Q))$$

Informell: Wenn die Bedingung B gilt, hat die Alternative die Semantik des THEN-Zweiges, andernfalls die Semantik des ELSE-Zweiges. Die beiden formalen Beschreibungen sind äquivalent.

Man beachte, daß B hier und im folgenden als *nebenwirkungsfreie* Funktion der Variablen (d.h. sie beeinflußt keine anderen Variablen des Programms) betrachtet wird.

Die bedingte Anweisung

Die Semantik der bedingten Anweisung ergibt sich, wenn die Definition der Semantik der leeren Anweisung herangezogen wird:

Semantik: $wp(\text{IF } B \text{ THEN } S \text{ END}, Q)$

$$\equiv wp(\text{IF } B \text{ THEN } S \text{ ELSE } \varepsilon \text{ END}, Q)$$

$$\equiv (B \Rightarrow wp(S, Q)) \wedge (\neg B \Rightarrow wp(\varepsilon, Q))$$

$$\equiv (B \Rightarrow wp(S, Q)) \wedge (\neg B \Rightarrow Q)$$

$$\equiv (B \wedge wp(S, Q)) \vee (\neg B \wedge Q)$$

Beispiele zur schwächsten Vorbedingung

1. Gegeben sei die folgende Anweisung S:

IF $x < 0$ THEN $x := -x$ ELSE $x := x-1$ END

mit $x \in \text{INTEGER}$ und die Nachbedingung $Q \equiv x \geq 0$.

Dann ergibt sich die schwächste Vorbedingung von S bezüglich der Nachbedingung Q zu

$$\begin{aligned}
 wp(S, Q) &\equiv ((x < 0) \wedge wp(x := -x, x \geq 0)) \\
 &\quad \vee ((x \geq 0) \wedge wp(x := x-1, x \geq 0)) \\
 &\equiv ((x < 0) \wedge (-x \geq 0)) \vee ((x \geq 0) \wedge (x-1 \geq 0)) \\
 &\equiv ((x < 0) \wedge (x \leq 0)) \vee ((x \geq 0) \wedge (x > 0)) \\
 &\equiv (x < 0) \vee (x > 0) \\
 &\equiv x \neq 0
 \end{aligned}$$

Die schwächste Vorbedingung, um nach Ausführung der Anweisung die Nachbedingung zu erfüllen, ist also $x \neq 0$. Eine Vorbedingung $x > 10$ wäre demnach hinreichend (weil dies $x \neq 0$ impliziert), $x < 10$ wäre es nicht.

2. Gesucht ist die schwächste Vorbedingung einer Anweisungssequenz S
 $t := t * x; i := i+1$

mit $i, n \in \text{INTEGER}$ und der Nachbedingung $Q \equiv i \leq n \wedge t = x^i$.

Dann gilt (mit der Nebenbedingung $x \neq 0$):

$$\begin{aligned}
 wp(S_1; S_2, Q) &\equiv wp(S_1, wp(S_2, Q)) \\
 &\equiv wp(t := t \cdot x, wp(i := i + 1, i \leq n \wedge t = x^i)) \\
 &\equiv wp(t := t \cdot x, i+1 \leq n \wedge t = x^{i+1}) \\
 &\equiv i+1 \leq n \wedge t \cdot x = x^{i+1} \\
 &\equiv i < n \wedge t = x^i
 \end{aligned}$$

Bemerkenswert an diesem Resultat ist, daß die Vorbedingung im zweiten Teil mit der Nachbedingung exakt übereinstimmt. Der Zusammenhang $t = x^i$ wird also durch die Anweisungssequenz S nicht verändert, obwohl sowohl t als auch i verändert werden. Man bezeichnet ein solches Prädikat als *Invariante*. Darauf wird später noch Bezug genommen.

4.2.3 Die Verifikation iterativer Programme

Ein iteratives Programm kann prinzipiell - i.a. aber relativ mühevoll - nach der gleichen Methode verifiziert werden. Das wesentliche Problem dabei ist aber die in der schwächsten Vorbedingung auftretende *Rekursion*. Im nachfolgenden P4.4a wird gezeigt, wie ein Programm mit den bisher vorgestellten Mitteln verifiziert werden kann.

Verifikation einer Iteration mit der schwächsten Vorbedingung

Gegeben ist das folgende Programm-Fragment F mit $x, n, t, i \in \text{INTEGER}$; x und n sind (hier) konstant.

```

WHILE i <= n DO                                     (* Programm P4.4a *)
  t := t * x;    i := i + 1;    (* Anweisungssequenz S *)
END (* WHILE *);

```

Damit soll x^n berechnet werden; die Nachbedingung lautet also $Q(t, i) \equiv t = x^n$.

Wir suchen nun nach der schwächsten Vorbedingung von F. Wird die Anweisungssequenz S (*Schleifenrumpf*) *nicht* durchlaufen, so lautet sie

$$wp_0(F, Q(t, i)) \equiv i > n \wedge Q(t, i)$$

(Durch $i > n$ ist dafür gesorgt, daß S nicht ausgeführt wird.)

Wird S *einmal* durchlaufen, so gilt:

$$wp_1(F, Q(t, i)) \equiv i \leq n \wedge wp(S, i > n \wedge Q(t, i))$$

und entsprechend für *zweimaliges* Durchlaufen

$$wp_2(F, Q(t, i)) \equiv i \leq n \wedge wp(S, i \leq n \wedge wp(S, i > n \wedge Q(t, i)))$$

Die Bedingung $i \leq n$ garantiert, daß die Schleife ausgeführt wird. Die Bedingung für den Abbruch muß *nach* der letzten Ausführung von S erfüllt sein und steht darum in der Nachbedingung des letzten (tiefstgeschachtelten) S.

Da die Schleife prinzipiell *beliebig* oft durchlaufen werden kann, ist die Menge der wp_i , die wir zu betrachten haben, nicht endlich. Daher gilt:

$$wp(F, Q(t, i)) \equiv wp_0(F, Q) \vee wp_1(F, Q) \vee wp_2(F, Q) \vee wp_3(F, Q) \vee \dots$$

Einsetzen von S ergibt (vgl. Wertzuweisungen):

$$\begin{aligned}
wp_1(F, Q(t, i)) &\equiv i \leq n \quad \wedge \quad wp(t := t \cdot x; i := i + 1, i > n \wedge Q(t, i)) \\
&\equiv i \leq n \quad \wedge \quad i + 1 > n \wedge Q(t \cdot x, i + 1) \\
&\equiv i = n \quad \wedge \quad Q(t \cdot x, i + 1)
\end{aligned}$$

$$\begin{aligned}
wp_2(F, Q(t, i)) &\equiv i \leq n \quad \wedge \quad wp(t := t \cdot x; i := i + 1, i \leq n \wedge \\
&\quad \quad \quad wp(t := t \cdot x; i := i + 1, i > n \wedge Q(t, i))) \\
&\equiv i \leq n \quad \wedge \quad wp(t := t \cdot x; i := i + 1, i = n \wedge Q(t \cdot x, i + 1)) \\
&\equiv i \leq n \quad \wedge \quad i + 1 = n \wedge Q(t \cdot x^2, i + 2) \\
&\equiv i + 1 = n \quad \wedge \quad Q(t \cdot x^2, i + 2)
\end{aligned}$$

Offensichtlich lautet damit die *allgemeine Form* für alle positiven ganzzahligen j :

$$\text{wp}_j (F, Q(t,i)) \quad \equiv \quad i+j-1 = n \wedge Q(t \cdot x^j, i+j)$$

Die Disjunktion aller wp_j läßt sich damit wie folgt ausdrücken:

$$\begin{aligned} \text{wp} (F, Q(t,i)) &\equiv \text{wp}_0 (F, Q) \vee \text{wp}_1 (F, Q) \vee \text{wp}_2 (F, Q) \vee \text{wp}_3 (F, Q) \vee \dots \\ &\equiv (i > n \wedge Q(t,i)) \vee \exists j \in \mathbb{N}: (i+j-1 = n \wedge Q(t \cdot x^j, i+j)) \end{aligned}$$

Damit haben wir eine *endliche* Beschreibung erreicht.

In diese setzen wir nun die Nachbedingung $Q(t,i) \equiv t = x^n$ ein und erhalten:

$$\begin{aligned} \text{wp} (F, Q(t,i)) &\equiv (i > n \wedge t = x^n) \vee \exists j \in \mathbb{N}: (i+j-1 = n \wedge t \cdot x^j = x^n) \\ &\equiv (i > n \wedge t = x^n) \vee (n+1-i > 0 \wedge t \cdot x^{n+1-i} = x^n) \\ &\equiv (i > n \wedge t = x^n) \vee (i \leq n \wedge (t = x^{i-1} \vee x = 0)) \end{aligned}$$

In Worten bedeutet dies: Entweder ist die Nachbedingung $t = x^n$ schon vor Eintritt in F erfüllt, dann wird durch $i > n$ garantiert, daß sie erfüllt bleibt, oder i ist nicht größer als n , und zu Beginn muß $t = x^{i-1}$ oder $x = 0$ gelten.

Dies ist die schwächste Vorbedingung; um sie zu erfüllen, suchen wir nun eine stärkere, die einfacher ist. Die erste Alternative ist in der Regel nicht zu erfüllen, denn sie erfordert (wegen $t = x^n$), daß wir schon *a priori* das Ergebnis kennen. Für spezielle Werte ($x = 1$ oder $x = 0$) ist dies der Fall, aber im allgemeinen ist es leichter, die zweite Alternative zu erfüllen. Wir wählen als Vorbedingung

$$P \quad \equiv \quad i = 1 \wedge t = 1 \wedge n \geq 0$$

Wie man sieht, gilt die Implikation $P \Rightarrow \text{wp} (F, Q(t,i))$, denn für $n = 0$ ist die erste Alternative, für $n > 0$ die zweite erfüllt. Um P zu garantieren, initialisieren wir i und t entsprechend; $n \geq 0$ muß durch den Kontext sichergestellt werden. Damit lautet das erweiterte Programm-Fragment zum Potenzieren (, wobei wir im Falle $x = 0$ für $n = 0$ das Ergebnis $0^0 = 1$ geliefert bekommen):

```
(* Vorbedingung: n >= 0 *)                                (* Programm P4.4b *)
i := 1;    t := 1;
WHILE i <= n DO
    t := t * x;    i := i+1;
END (* WHILE *);
(* Nachbedingung: t = x hoch n *)
```

Will man ein *robustes* Programm schreiben, so muß man die Vorbedingung auf TRUE reduzieren (d.h. es kann nichts mehr "schiefgehen"). Hier ist dies durch eine entsprechende Eingangsprüfung möglich:

```

(* Vorbedingung TRUE *)                                (* Programm P4.4c *)
IF n >= 0 THEN
  (* n>=0 ist sicher, P4.4b *) ...
ELSE
  (* Fehlermeldung *) ...
END (* IF *)

```

Partielle und totale Korrektheit

Wir haben gesehen, daß bei Einhaltung der Vorbedingung(en) nie ein falsches Ergebnis entsteht (*partielle Korrektheit*). Das ist nicht ausreichend, denn wir wollen sicherstellen, daß wir stets das richtige Ergebnis erhalten (*totale Korrektheit*). Dafür ist es notwendig, das *Terminieren* zu beweisen.

Dieser Nachweis ist im Programm P4.4a trivial: Wir müssen zeigen, daß die Schleife nicht oder nur endlich oft ausgeführt wird, und daß der Inhalt der Schleife selbst stets nur endlich viele Schritte erfordert. Für $n = 0$ gilt das erste, für $n > 0$ das zweite (n Durchläufe), wobei jeder Durchlauf aus zwei arithmetischen Operationen und zwei Wertzuweisungen besteht. Daß nicht in jedem Falle aus der partiellen die totale Korrektheit folgt, zeigt das folgende einfache Beispiel:

```

P ≡ k ∈ N ∧ a ∈ N
{ WHILE (k MOD a) > 0 DO k := (k+2) MOD a END }
Q ≡ k MOD a = 0

```

Wie man sieht, wird die Iteration nur abgebrochen, wenn die Nachbedingung erfüllt ist. Sind aber zu Beginn k ungerade und a gerade, so endet die Iteration nie, das Programm terminiert nicht. Es ist darum nur partiell korrekt.

Der Schleifenrumpf als Variablen-Transformator

Gegeben sei die Schleife

```
WHILE B(V) DO S END
```

Darin ist V der *Vektor aller Variablen*, die von S verändert werden, B eine von V abhängige *Wiederholbedingung*. Wir beschreiben die Änderung von V bei *einer* Ausführung von S bei insgesamt k Variablen durch

$$V = h(V') = [h_1(V'), h_2(V'), \dots, h_k(V')]$$

wobei h der durch den Schleifenrumpf realisierte *Variablen-Transformator* ist. Im Programm P4.4a besteht V aus t und i (x ist konstant). Bei jeder Ausführung von S wird t durch $t \cdot x$ ersetzt, i durch $i+1$.

Damit gilt: $h(t, i) = [t \cdot x, i+1]$.

j Ausführungen der Schleife entsprechen j Anwendungen von h :

$$V_j = h(V_{j-1}) = h(h(V_{j-2})) = h(h(h(\dots h(V_0) \dots))) = H(V_0, j), \text{ wobei}$$

V_0 der Variablen-Vektor nach 0 Durchläufen, also vor der erstmaligen Ausführung der Schleife ist,

$H(V_0, j)$ vereinfachend die j -malige Anwendung von h auf V_0 beschreibt, mit dem Sonderfall $j = 0$ (identische Abbildung $H(V_0, 0) = V_0$).

Im Programm P4.4a gilt $H(t, i, j) = [t \cdot x^j, i+j]$.

Beispiel: Maximales Element eines Feldes finden

Gegeben ist das folgende Programm-Fragment; A sei indiziert mit 1 bis n , $n \geq 1$.

```

WHILE B(i) DO                                (* Programm P4.5a *)
  IF A[i] > x THEN  x := A[i];  END;
  i := i + 1;
END (* WHILE *);

```

Die Wiederholbedingung ist hier allgemein als Funktion von i beschrieben:

$$h(x, i) = [\max(x, A[i]), i+1]$$

Bei jedem Durchlauf wird x also durch das Maximum von x und $A[i]$ und i durch $i+1$ ersetzt. Für j Durchläufe folgt:

$$H(x, i, j) = [\max(x, A[i], A[i+1], A[i+2], \dots, A[i+j-1]), i+j].$$

Sei nun die Nachbedingung

$$Q(x, i) \equiv x = \max(A[1], A[2], \dots, A[n]).$$

Wie im Programm P4.4a nehmen wir wieder an, daß diese Nachbedingung nach j Durchläufen erfüllt ist, und folgern:

$$\max(x, A[i], \dots, A[i+j-1]) = \max(A[1], A[2], \dots, A[n]).$$

Natürlich gibt es auch hier wieder verschiedene Vorbedingungen, um diese Bedingung zu erfüllen; eine einfache, für den allgemeinen Fall gültige ergibt sich, wenn wir die Argumente in der Klammer paarweise gleichsetzen:

$$(x = A[1] \wedge n = 1) \vee (i = 2 \wedge i+j-1 = n \wedge j \geq 1).$$

Tatsächlich können wir auch $j = 0$ zulassen, denn für $n = 1$ wird der Anfangswert von x gleichzeitig sein Endwert (d.h. das einzige Element des Feldes ist zugleich sein größtes). Wir eliminieren $j = n-1$ und erhalten:

$$x = A[1] \wedge i = 2 \wedge n > 0.$$

B(i) muß nun so gewählt werden, daß die Schleife genau j mal durchlaufen wird, daß also gilt

$$B(i+j) \equiv B(n+1) \equiv \text{FALSE}$$

$$B(i+j-1) \equiv B(n) \equiv \text{TRUE}$$

Offensichtlich ist $B(i) \equiv i \leq n$ eine geeignete Funktion von i.

Wir haben also ermittelt: Mit der Anfangsbedingung $x = A[1] \wedge i = 2 \wedge n > 0$ und der Wiederholbedingung $i \leq n$ bei global gegebenem n realisiert FMax eine Funktion, die in x den Wert des maximalen Elementes von Feld A liefert.

```
(* Programm P4.5b *)
```

```

PROCEDURE FMax (A : TestfeldTyp) : REAL;
VAR x : REAL;          (* Vertreter von FMax in der Funktion *)
    i : [1..Feldlaenge];
BEGIN (* FMax *)
    x := A[1];    i := 2;
    WHILE i <= n DO
        IF A[i] > x THEN x := A[i];  END;
        i := i + 1;
    END (* WHILE *);
    RETURN x;
END FMax;

```

Verifikation verschiedener Schleifenformen

Die bisherigen Konzepte für die Schleife mit vorangehender Prüfung, die WHILE-Schleife also, lassen sich auf die übrigen Schleifenarten übertragen.

Die Lauf-Schleife

Die Lauf- oder FOR-Schleife kann durch eine Transformation als WHILE-Schleife dargestellt werden. Das Problem des Terminierens besteht bei der FOR-Schleife nicht, die Syntax garantiert das Terminieren, wenn der Rumpf der Schleife terminiert.

Im Programm P4.4a sollte statt

```

i := 1;    t := 1;
WHILE i <= n DO t := t * x;  i := i+1;  END

```

besser formuliert werden:

```

t := 1;
FOR i := 1 TO n DO t := t * x;  END

```

Im Programm P4.5b ist besser als

```

i := 2;
WHILE i <= n DO IF A[i] > x THEN x := A[i] END;    i := i + 1;
END;

```

die äquivalente, aber verständlichere FOR-Schleife:

```
x := A [1];
FOR i := 2 TO n DO
  IF A[i] > x THEN x := A[i]; END;
END (* FOR *);
```

Die Schleife mit nachfolgender Prüfung

Die Schleife

```
REPEAT S UNTIL B
```

ist gleichbedeutend mit

```
S; WHILE NOT B DO S END
```

Die Schleife ohne Wiederholungsprüfung

Die Schleife ohne Wiederholungsprüfung (LOOP) läßt sich nur in einfachen Fällen schematisch in eine WHILE-Schleife überführen. Beispielsweise gilt:

```
wp (LOOP S1; IF B THEN EXIT END; S2 END, Q)
  ≡ wp (S1; WHILE NOT B DO S2; S1 END, Q).
```

Bei mehreren oder in geschachtelten Anweisungen auftretenden EXIT-Anweisungen ist eine kompliziertere Umsetzung notwendig, auf die hier nicht näher eingegangen werden soll. Man beachte aber allgemein den Grundsatz, daß alles, was schwer zu verifizieren ist (beispielsweise eine Schleife mit mehreren Ausgängen), auch schwer verständlich und daher fehlerträchtig ist.

4.2.4 Beschreibung einer Schleife durch eine Invariante

In diesem Abschnitt wird ein anderer Ansatz für die Verifikation eines iterativen Programms gezeigt, der sich wesentlich leichter anwenden läßt.

Das Prinzip der Invariante

Wir haben bereits *Transformatoren* $h(V)$ und $H(V, j)$ eingeführt, um die Semantik des Schleifenrumpfes zu beschreiben. Eine (*Schleifen-*) *Invariante* INV ist eine Aussage über V , die *stets* gültig ist, wenn die Wiederholbedingung geprüft wird. Es gilt also

$$\text{INV}(V_0) \equiv \text{TRUE},$$

außerdem unter der Bedingung $B(V_0) \wedge B(V_1) \wedge \dots \wedge B(V_m)$ auch

$$\text{INV}(h(V_m)) \equiv \text{INV}(V_{m+1}) \equiv \text{TRUE}.$$

Natürlich ist " $\equiv \text{TRUE}$ " überflüssig; wir können die Invariante also charakterisieren durch

$$\text{INV}(V_0) \wedge B(V_0) \wedge B(V_1) \wedge \dots \wedge B(V_m) \Rightarrow \text{INV}(h(V_m)).$$

Die Invariante beschreibt also einen *Zusammenhang der Variablen*, der erhalten bleibt, solange die Schleife durchlaufen wird. Beim *Abbruch* der Iteration ist $B(V)$ nicht mehr gegeben. Damit gilt an diesem Punkt:

$$\text{INV}(V_j) \wedge \neg B(V_j).$$

Bei richtiger Wahl der Invariante läßt sich daraus die Nachbedingung Q ableiten:

$$(\text{INV}(V_j) \wedge \neg B(V_j)) \Rightarrow Q(V_j).$$

Wir sehen hier, daß eine geeignete Invariante drei verschiedene Bedingungen erfüllen muß:

- Sie muß zu Beginn gelten.
- Sie muß, sofern B erfüllt war, unter Anwendung von h tatsächlich invariant sein.
- Sie muß hinreichend aussagekräftig sein, um nach Abbruch den Schluß auf Q zu gestatten.

Das letzte Kriterium wird z.B. nicht erfüllt von der Invariante $\text{INV}(V) \equiv \text{TRUE}$, die natürlich eine Invariante jeder Schleife ist.

Beispiel:

Im Programm P4.4a ist die Invariante

$$\text{INV}(i, t) \equiv i \leq n+1 \wedge t = x^{i-1}.$$

Dies läßt sich durch *Induktion* beweisen:

Seien die Invariante und die Wiederholbedingung gültig für i_m und t_m , die Werte von i und m nach dem m -ten Schleifendurchlauf. Dann gilt also

$$\text{INV}(i_m, t_m) \wedge B(i_m) \equiv i_m \leq n+1 \wedge t_m = x_m^{i_m-1} \wedge i_m \leq n.$$

Wegen $h(t, i) = [t \cdot x, i+1]$ gilt:

$$t_{m+1} = t_m \cdot x = x_m^{i_m-1} \cdot x = x_m^{i_m} \wedge i_{m+1} = i_m + 1$$

also nach Übergang auf i_{m+1} :

$$t_{m+1} = x_{m+1}^{i_{m+1}-1} \wedge i_{m+1} \leq n+1 \equiv \text{INV}(i_{m+1}, t_{m+1}),$$

was zu beweisen war.

Dafür, daß die Invariante am Anfang $x_{i_m-1}^m$ tatsächlich erfüllt ist, ist eine hinreichende Vorbedingung

$$P(i, t, n) \equiv [i = 1 \wedge n \geq 0 \wedge t = 1] \Rightarrow INV(i, t).$$

Mit der Wiederholbedingung $B(i) \equiv i \leq n$ folgt als Nachbedingung

$$\begin{aligned} INV(V) \wedge \neg B(V) &\equiv \\ i \leq n+1 \wedge t = x^{i-1} \wedge i > n &\equiv i = n+1 \wedge t = x^{i-1} \Rightarrow t = x^n \equiv Q(i, t). \end{aligned}$$

Im Programm P4.5 kann analog gezeigt werden:

$$\begin{aligned} INV(i, x) &\equiv 2 \leq i \leq n+1 \wedge x = \max(A[1], A[2], \dots, A[i-1]) \\ P(i, x, n) &\equiv n > 0 \wedge i = 2 \wedge x = A[1] \Rightarrow INV(i, x) \\ B(i, x) &\equiv i \leq n. \end{aligned}$$

Damit folgt für den Abbruch der Iteration:

$$\begin{aligned} INV(i, x) \wedge \neg B(i, x) & \\ \equiv 2 \leq i \leq n+1 \wedge x = \max(A[1], A[2], \dots, A[i-1]) \wedge i > n & \\ \equiv i = n+1 \wedge x = \max(A[1], A[2], \dots, A[i-1]) & \\ \Rightarrow x = \max(A[1], A[2], \dots, A[n]). & \end{aligned}$$

4.2.5 Konstruktion iterativer Programme

Die bisher gezeigte Anwendung der Invarianten zur Verifikation eines Programms *a posteriori* ist in der Praxis nicht sinnvoll. Dort geht es vielmehr darum, Schleifen korrekt zu *konstruieren*. Das bei dieser Schleifenkonstruktion anzuwendende Verfahren wird durch das *Schema* in Abb. 4.1 deutlich, das die zur Konstruktion einer Schleife durchzuführenden sechs Schritte zusammenfaßt.

1. Festlegung der *Nachbedingung* Q (als Ziel der Konstruktion).
2. "Entdeckung" einer geeigneten *Invariante* INV.

Dies ist ein schöpferischer Akt! Er besteht darin, eine Zerlegung des Problems in Teilprobleme zu finden, deren Lösung jeweils die Invariante erfüllt.
3. Festlegung der *Vorbedingung* P und Ableitung der notwendigen *Initialisierung* (Eingangsprüfungen oder Anforderungen an vorangehende Programmteile), d.h. *Nachweis*, daß P die Invariante impliziert.
4. Festlegung der *Wiederholbedingung* B.

5. *Nachweis*, daß die in 2. bestimmte Invariante wirklich eine Invariante ist, d.h. bei gültiger Wiederholbedingung auch nach Durchlaufen des Schleifenrumpfes weiterhin gilt.
6. *Nachweis*, daß bei nicht mehr gültiger Wiederholbedingung die Nachbedingung Q gilt.

Ist Schritt 6 nicht erfolgreich, so wiederholt man den Prozeß ab Schritt 2.

Abb. 4.1 Schrittweise Konstruktion einer Schleife

Nachfolgend soll die Lösung zweier Probleme (Fakultät, Mittelwertbildung) jeweils durch Konstruktion einer Schleife anhand einer Invariante gezeigt werden.

Beispiel A: Iteratives Fakultät-Programm

(die Numerierung verweist auf die notwendigen sechs Schritte)

1. $Q \equiv \text{Resultat} = n !$
2. $INV \equiv \text{Resultat} = i ! \wedge 0 \leq i \leq n$

Implementierung des Rumpfes S:

$i := i+1; \text{Resultat} := \text{Resultat} * i$

$$\begin{aligned} wp(S, Q) &\equiv wp(i := i+1, wp(\text{Resultat} := \text{Resultat} \cdot i, Q(i, \text{Resultat}))) \\ &\equiv wp(i := i+1, Q(i, \text{Resultat} \cdot i)) \equiv Q(i+1, \text{Resultat} \cdot (i+1)) \end{aligned}$$

Daraus folgt die Transformation $h(i, \text{Resultat}) = [i+1, \text{Resultat} \cdot (i+1)]$, die (zusammen mit der noch zu definierenden Wiederholbedingung B) die Invariante garantiert. Man beachte, daß S bei umgekehrter Reihenfolge falsch wäre, denn $h(V)$ verletzt dann die Invariante.

3. $P \equiv \text{Resultat} = 1 \wedge i = 0 \wedge n \geq 0$
 $P \Rightarrow INV ?$
 Beweis: $\text{Resultat} = 1 = i ! \wedge 0 = i \leq n$

4. $B \equiv i < n$

5. $INV(V) \wedge B(V) \Rightarrow INV(V') ?$
 Beweis: $\text{Resultat} = i ! \wedge 0 \leq i \leq n \wedge i < n$
 $\Rightarrow \text{Resultat} \cdot (i+1) = \text{Resultat}' = i ! \cdot (i+1) = i' ! \wedge i+1 = i' \leq n$

6. $INV(V) \wedge \neg B(V) \Rightarrow Q(V)$?

Beweis: $Resultat = i! \wedge 0 \leq i \leq n \wedge i \geq n$
 $\Rightarrow Resultat = i! \wedge i = n \Rightarrow Resultat = n!$

Damit ist das Programm erfolgreich konstruiert. Natürlich ist es hier sinnvoll, die Einschränkung $n \geq 0$ in der Vorbedingung P von Anfang an zu machen, denn für negative Argumente ist die Fakultät nicht definiert. Die Einschränkung ergibt sich aber in jedem Fall, wenn man versucht, $P \Rightarrow INV$ zu beweisen.

Wir können nun die komplette *Implementierung* zusammenstellen:

```
IF n >= 0 THEN
  Resultat := 1;   i := 0;
  WHILE i < n DO  i := i+1;  Resultat := Resultat * i;  END;
ELSE
  (* Fehlermeldung *) ...
END (* IF *);
```

In diesem Fall ist eine FOR-Schleife angebracht. Wir müssen dazu die *Inkrementierung* von i ans Ende des Schleifenrumpfes bringen:

```
WHILE i < n DO
  Resultat := Resultat * (i+1);  i := i+1;
END (* WHILE *);
```

Nun ersetzen wir noch zur Vereinfachung $i+1$ durch j (oder i durch $j-1$) und erhalten nach trivialen Vereinfachungen:

```
Resultat := 1;  j := 1;
WHILE j <= n DO  Resultat := Resultat * j;  j := j+1;  END;
```

Hierfür kann nun die Laufschleife geschrieben werden:

```
Resultat := 1;
FOR j := 1 TO n DO  Resultat := Resultat * j;  END;
```

Bei dieser Lösung fällt ins Auge, daß es eine etwas effizientere Lösung gibt, bei der j nicht von 1, sondern von 2 an läuft. Die entsprechende Initialisierung mit

$P \equiv Resultat = 1 \wedge i = 1 \wedge n \geq 0$

impliziert aber nicht mehr die Invariante. Um sie zu verifizieren, benötigen wir eine kompliziertere Invariante, in der der Fall, daß die Lösung schon zu Beginn in $Resultat$ steht, separat behandelt ist:

$INV \equiv (Resultat = n! \wedge i \geq n) \vee (Resultat = i! \wedge 0 \leq i \leq n).$

Implementierung mit FOR-Schleife:

```
Resultat := 1;
FOR j := 2 TO n DO Resultat := Resultat * j; END;
```

Als Prozedur sieht die Fakultät-Funktion damit wie folgt aus (mit Umbenennung von Resultat):

```
(* Iterative Berechnung der Fakultät; *) (* Programm P4.6 *)
(* Der Typ ArgBereich ist global.      *)

PROCEDURE Fakultät (Argument: ArgBereich): CARDINAL;
VAR Produkt : CARDINAL; (* Vertreter von Fakultät in der
                          Funktion *)
    Index    : ArgBereich;
BEGIN (* Fakultät *)
    Produkt := 1;
    FOR Index := 2 TO Argument DO
        Produkt := Produkt * Index;
    END (* FOR *);
    RETURN Produkt;
END Fakultät;
```

Beispiel B: Mittelwert über die positive Sequenz eines Arrays

In diesem Beispiel soll ein Programm konstruiert werden, das feststellt, wieviele aufeinanderfolgende Elemente eines Arrays (ab Index 1) positiv sind, und zusätzlich den Mittelwert dieser Elemente liefert. Ist die Zahl der positiven Elemente null, so hat der Mittelwert einen vordefinierten Wert. Die Behandlung dieses Beispiels erfolgt weniger detailliert als beim Programm P4.6, es soll vor allem das Prinzip angedeutet werden.

Das Feld sei F : $\text{ARRAY } [1..\text{Lng}F] \text{ OF REAL}$ (mit $\text{Lng}F \geq 1$); es wird unterstellt, daß alle Elemente des Arrays definierte Werte haben. Die Probleme des Überlaufs und der Rundungsfehler bei der REAL-Addition werden ignoriert.

Im 1. Schritt legen wir wieder die bereits umgangssprachlich formulierte *Nachbedingung* präzise fest. Gesucht ist k , so daß gilt:

$$\begin{aligned} Q &\equiv 0 \leq k \leq \text{Lng}F \\ &\wedge \forall j \in \{1 \dots k\}: F[j] > 0 \wedge (F[k+1] \leq 0 \vee k = \text{Lng}F) \\ &\wedge (k = 0 \vee \text{Mittelwert} = \frac{1}{k} \sum_{i=1}^k F[i]) \end{aligned}$$

Man beachte die Bedingung $k = \text{Lng}F$, die den Fall abdeckt, in dem alle Elemente des Feldes positiv sind. Dann ist die Bezugnahme auf $F[k+1]$ nicht möglich.

Wir bilden nun in den Schritten 2-4 wieder *Invariante*, *Initialisierung* und *Wiederholbedingung*:

$$\begin{aligned} \text{INV} &\equiv 0 \leq k \leq \text{LngF} \\ &\wedge \quad \forall j \in \{1 \dots k\}: F[j] > 0 \\ &\wedge \quad (k = 0 \vee \text{Mittelwert} = \frac{1}{k} \sum_{i=1}^k F[i]) \end{aligned}$$

Damit zeigt $k+1$ stets auf das zu prüfende Element des Feldes oder auf ein fiktives hinter dem Ende. Implementierung:

```
Mittelwert := (Mittelwert * FLOAT(k) + F[k+1]) / FLOAT(k+1);
k := k+1;
```

$P \equiv k = 0$ (Implementierung: $k := 0$)

$B \equiv F[k+1] > 0 \wedge k < \text{LngF}$

Implementierung: Die naive Umsetzung in die Bedingung “ $(F[k+1] > 0.0) \text{ AND } (k < \text{LngF})$ ” ist logisch korrekt, ergibt jedoch ein falsches Programm, da sie nach den Regeln der Sprache MODULA-2 von links nach rechts bearbeitet wird und daher am Ende des Feldes zur Auswertung eines undefinierten Ausdrucks “ $F[\text{LngF}+1]$ ”, also zu einem Laufzeitfehler führt.

Bei undefinierten Teilausdrücken sind die logischen Operatoren im Programm *nicht* kommutativ, denn “TRUE OR undef.” hat den Wert TRUE, “FALSE AND undef.” den Wert FALSE, wogegen “undef. OR TRUE” und “undef. AND FALSE” undefiniert sind! Wir implementieren also B korrekt durch “ $(k < \text{LngF}) \text{ AND } (F[k+1] > 0.0)$ ”.

Daß die Invariante bei gültiger Wiederholbedingung erhalten bleibt (5. Schritt), ist unmittelbar einsichtig. Die *Nachbedingung* geht wie in Programm P4.6 aus INV und $\neg B$ hervor (Schritt 6).

Das Programm P4.7 lautet damit insgesamt:

```
k := 0;                                     (* Programm P4.7a *)
WHILE (k < LngF) AND (F[k+1] > 0.0) DO
  Mittelwert := (Mittelwert * FLOAT(k) + F[k+1]) / FLOAT(k+1);
  k := k+1;
END (* WHILE *);
```

Der Rumpf der Schleife sieht eleganter aus, wenn man die Inkrementierung von k vorzieht:

```

k := k+1;
Mittelwert := (Mittelwert * FLOAT(k-1) + F[k]) / FLOAT(k);

```

Obwohl dieses Programm verifiziert werden kann, hat es zwei *Mängel*:

- Es ist formal unschön, daß "Mittelwert" zu Beginn undefiniert ist.
- Die laufende Multiplikation und Division ist ineffizient und verstärkt den Einfluß von Rundungsfehlern. Darum ist es praktisch wesentlich besser, die Division nur einmal hinter der Schleife durchzuführen. Die Initialisierung der Zwischensumme ist dann zwingend notwendig.

Eine in diesem Sinne verbesserte Programmvariante ist:

```

k := 0;   Summe := 0.0;                                (* Programm P4.7b *)
WHILE (k < LngF) AND (F[k+1] > 0.0) DO
  k := k+1;   Summe := Summe + F[k];
END (* WHILE *);
IF k > 0 THEN
  Mittelwert := Summe / FLOAT(k);
ELSE
  Mittelwert := 0.0;
END (* IF *);

```

Natürlich gelten für dieses Programm jeweils eine andere Invariante, Vor- und Nachbedingung.

Eine FOR-Schleife kommt hier kaum in Frage, sie hätte gegenüber der Lösung oben nur Nachteile. Anwendbar ist hingegen die REPEAT-Schleife, denn eine Abfrage, ob das Feld mindestens ein Element enthält, ist in jedem Falle nötig. In P4.7c ist der Algorithmus als Funktion formuliert.

```

PROCEDURE Mittelwert () : REAL;                          (* Programm P4.7c *)
VAR k : INTEGER;   Summe : REAL;
BEGIN (* Mittelwert *)
  IF F[1] > 0.0 THEN
    k := 0;   Summe := 0.0;
    REPEAT
      k := k+1;   Summe := Summe + F[k];
    UNTIL (k = LngF) OR (F[k+1] <= 0.0);
    RETURN Summe / FLOAT(k);
  ELSE
    RETURN 0.0;
  END (* IF *);
END Mittelwert;

```


4.2.6 Zusammenfassung

Die Beschreibung der Semantik kann u.a. folgenden Zwecken dienen:

- Axiomatische *Definition* einer Programmiersprache
- *Programmbeschreibung* (Was tut das Programm?)
- *Programmspezifikation* (Was soll das Programm tun?)
- *Programmverifikation* (Tut das Programm das, was es soll?).

Die *Definition* ist die Vorschrift für denjenigen, der ein Sprachsystem (Interpreter, Compiler) implementiert; ein Programmierer wird in der Regel unterstellen, daß die Implementierung der Sprache den Axiomen genügt.

Eine *Spezifikation* ist eine Präzisierung der Anforderungen an ein Programm ("Pflichtenheft").

Die *Beschreibung* des Programms entsteht dann durch Zusammensetzung der Axiome entsprechend dem Programmaufbau. Sie setzt also die formale Definition der Programmiersprache voraus.

Die *Verifikation* setzt voraus, daß eine Spezifikation und eine Beschreibung aufgestellt wurden. Sie besteht im Nachweis, daß Beschreibung und Spezifikation verträglich sind.

4.3 Test

4.3.1 Begriffsbildungen und Prinzipien

Formale Beweise sind - wie sie bei der Verifikation von Programmen geführt werden - im allgemeinen sehr aufwendig und heutzutage in der Praxis nur für kleinere Programme durchführbar. Weiterhin sind vielen Programmierern die notwendigen Semantikdefinitionen und die formalen Beweistechniken gar nicht bekannt. Daher führt man meist Tests durch, mit denen man hofft, offensichtliche Fehler und Schwachstellen zu entdecken.

Wie in der Einleitung des Kapitels 4 gesagt wurde, zielt ein Test darauf, *Fehler zu entdecken*; Testen heißt also: Spuren suchen. Ein Test ist in dem Sinne erfolgreich, wenn eine Spur, das Symptom eines Fehlers, entdeckt wurde. War dies der Fall, wurde also eine Spur gefunden, so folgt die Spurenverfolgung (*Fehlersuche*) und schließlich das Einfangen des unerwünschten Eindringlings (*Fehlerbehebung*).

Die beiden letzten Tätigkeiten, die oft im Wort "Debugging" (engl. "Entwanzen") zusammengefaßt sind, gehören aber nicht zum Test und sind von diesem aus gutem Grunde strikt zu trennen: Auch der Arzt beginnt nicht sofort mit der Therapie, wenn er das erste Symptom bemerkt hat, sondern führt seine Untersuchung zunächst systematisch zu Ende. Andernfalls könnten die weiteren Untersuchungsergebnisse durch die Therapie verfälscht werden; vielleicht zeigen die weiteren Resultate auch, daß eine ganz andere Therapie angezeigt ist.

Dieser Abschnitt befaßt sich also mit dem Test im engeren Sinne, mit der Suche nach Hinweisen auf Fehler. Ein Programm oder Programmteil enthält einen *Fehler*, wenn es

- nicht das tut, was es soll,
- etwas tut, was es nicht soll.

Die Erfolge dessen, der testet, sind die Mißerfolge dessen, der das Programm entwickelt hat, und umgekehrt. Für den Programmierer ist ein Programmlauf mit einer bestimmten Eingabe erfolgreich, wenn das Programm korrekt arbeitet. Umgekehrt ist der Programmlauf für den Tester erfolgreich, wenn das Programm mit dieser Eingabe fehlerhaft arbeitet.

Obwohl beide, Tester und Programmierer, temporär, d.h. bei einem Programmlauf, einander widerstrebende Ziele verfolgen, ist das Fernziel für beide identisch: die Verbesserung des Programms und der Nachweis seiner Korrektheit. Diese Beobachtung hat Konsequenzen für die Durchführung des Testens: Da der Programmierer in

erster Linie daran interessiert ist, *keine* Fehler zu finden, sollte der Test möglichst von einer fremden Person durchgeführt werden. Bei der Arbeit in Gruppen testen die Mitglieder ihre Programmteile gegenseitig. Auch hier gilt das Prinzip des *egoless programming*, d.h., Ziel ist die Korrektheit des Gesamtsystems und nicht die Abgrenzung und das "Verstecken" des eigenen Programms. Es geht nicht darum, wer in welchem Programmteil einen Fehler programmiert hat.

Tests werden im Zuge der Software-Entwicklung in verschiedenen Stadien durchgeführt: Derjenige, der ein Programm (oder eine Komponente) fertiggestellt hat, wird dieses selbst "ausprobieren", also feststellen, ob es überhaupt plausible Ergebnisse liefert. Ein Test im Sinne der Definition aber kann erst beginnen, wenn das Programm stabil ist, also nicht mehr "abstürzt" oder offensichtlich "Unsinn" produziert.

Wir unterscheiden mindestens folgende *Korrektheitsebenen*:

1. Ebene: Das Programm "läuft", d.h. es entspricht der Syntax der gewählten Programmiersprache und wird vom Rechner nicht abgewiesen. Für die gewählte Eingabe läuft das Programm bis zum Ende durch. Diese Ebene erreicht man quasi automatisch: Der Rechner testet auf jeden Fall Programme auf ihre *syntaktische Korrektheit*.
2. Ebene: Das Programm liefert für *einige* ausgewählte Eingabewerte korrekte Ergebnisse. Typisch sind die Eingaben, die in der Einsatzumgebung überwiegend auftreten.
3. Ebene: Das Programm liefert für *irgendwelche* Eingaben (zufällig erzeugte oder "Spielbeispiele") eine korrekte Ausgabe.
4. Ebene: Das Programm liefert für *alle* zulässigen Eingabewerte (auch boshaft gewählte) korrekte Ergebnisse.

Der "richtige" Test wird - als eigens geplante und budgetierte Aktivität - von anderen Leuten durchgeführt, nicht von den Entwicklern, denn fast niemand bringt es fertig, sein eigenes Werk als fehlerhaft zu überführen. Auch wird der Entwickler sehr wahrscheinlich beim Test von denselben Irrtümern und Mißverständnissen ausgehen wie bei der Implementierung. Eine solche Testphase ist wie die übrigen Phasen außerhalb der Codierung Gegenstand des Software Engineerings. Nachfolgend sollen einige technische Hinweise zum Test weitergegeben werden, wie sie - als Gegenstück und Ergänzung der Verifikation - auch vom einzelnen Entwickler mit Vorteil angewandt werden können.

Für einen Test sind erforderlich:

- das zu testende Programm (konkret seine Implementierung),
- Testdaten, bestehend aus einer Testmenge von Eingabedaten und Soll-Resultaten.

Die Spezifikation ist für beides Voraussetzung und damit - wie bei der Verifikation - der Dreh- und Angelpunkt. Natürlich wird in der Praxis nur selten eine *formale*

Spezifikation vorliegen. Aber auch eine informelle Definition der Aufgabe ist ein viel besserer Ausgangspunkt für die Prüfung als die vage Vorstellung, was das Programm leisten soll.

Die Prüfung der *Ist-Resultate*, d.h. der vom Programm bezüglich aller Eingabedaten gelieferten Ergebnisse, gegen die *Soll-Resultate* leistet am sichersten ein (auf den meisten Rechnern verfügbares) *Vergleichsprogramm*. Bei der reinen Sichtprüfung ist akribische Sorgfalt nötig, denn nur wenige Fehler springen ins Auge.

Alle Rahmenbedingungen des Tests sowie die Testdaten und -resultate sind präzise zu *dokumentieren*. Andernfalls kann es passieren, daß man tagelang nach einem Fehler sucht, der in Wirklichkeit in der neuen Betriebssystem- oder Compilerversion liegt oder durch das Anbinden der falschen Bibliothek entstanden ist. Bei späteren Tests - etwa nach Programm-Modifikationen - kann man auch mit Vorteil auf die alten Resultate zurückgreifen (*Regressionstest*), sie dienen dann als Soll-Resultate.

Viele Probleme beim Test rühren daher, daß keine Spezifikation vorliegt und die Soll-Resultate nicht *vor* dem Test ermittelt worden sind. Wir halten also fest: Eine Programmausführung ohne Soll-Resultate ist ein Probelauf, aber kein Test. Die Hoffnung, daß man es dem Ergebnis schon ansehen werde, ob es stimmt, ist unbegründet.

4.3.2 Grenzen des Tests

Ist die formale Verifikation ein "pessimistisches" Verfahren ("Ich halte das Programm erst für korrekt, wenn es verifiziert ist"), so erscheint der Test als "optimistischer" Ansatz ("Wenn ich keinen Fehler finde, ist das Programm vermutlich korrekt"). Dieser Schluß ist logisch falsch, er beruht auf einer unzulässigen Umkehrung der Implikation: Sei F die Aussage, daß ein Fehler entdeckt wurde, und K die Aussage, daß das Programm korrekt ist. Offenbar gilt:

$$F \Rightarrow \neg K$$

Wenn man Voraussetzung und Folgerung negiert, muß man die Implikation herumdrehen:

$$\neg F \Leftarrow K$$

Das ist die banale Aussage, daß in einem korrekten Programm keine Fehler zu finden sind. Die angestrebte Aussage, daß nämlich aus $\neg F$ ein Schluß auf K möglich ist, setzt voraus, daß wir aus der Implikation oben eine Äquivalenz machen:

$$F \Leftrightarrow \neg K$$

Da wir spezielle Ergebnisse nicht verallgemeinern dürfen, müssen dazu *alle möglichen Eingabekombinationen* getestet worden sein. Dies ist praktisch nur in sehr

wenigen, extrem einfachen Fällen möglich, beispielsweise bei einer Funktion mit einem einzigen ganzzahligen Parameter, etwa bei der Funktion ODD (n), die uns sagt, ob eine Zahl ungerade ist. Wir durchlaufen zum Test den (positiven und negativen) Zahlbereich zweimal in Zweierschritten, einmal ab 0, einmal ab 1. Im ersten Fall muß das Resultat stets FALSE, im zweiten TRUE sein.

Schon bei zwei Parametern ist ein vollständiger Test praktisch ausgeschlossen: Kann (bei 32 bit Wortlänge) jede der Zahlen 2^{32} verschiedene Werte annehmen, so gibt es insgesamt $2^{64} > 10^{19}$ Kombinationen. Beachtet man, daß drei Jahre etwa 10^8 s haben, dann wird die Aussichtslosigkeit des Unterfangens deutlich.

Gäbe es eine Steigerung von "aussichtslos", so ist sie nötig, um die Anforderung zu charakterisieren, daß die zu prüfende Komponente *nichts* tut, was sie *nicht* tun soll, also beispielsweise nicht Daten eines anderen Programms verfälscht. Im konkreten Fall (und bei vorsätzlicher Fehlprogrammierung) kann das bedeuten, daß sich bei einer bestimmten Kombination der Eingabewerte z.B. einer Hochregal-Steuerung das (von einem ganz anderen System verwaltete) Gehalt eines bestimmten Mitarbeiters verdoppelt. Wie man sieht, hilft hier der Test nichts mehr, nur konstruktive Maßnahmen geben bescheidene Sicherheit (z.B. darf die Lohnbuchhaltung einfach technisch nicht von der Hochregal-Steuerung aus zugänglich sein).

In der Regel ist eine *konstruktive Lösung*, wo immer sie in Frage kommt, dem Test vorzuziehen. So können bestimmte logische Beziehungen (Prädikate wie Vor- und Nachbedingungen, Invarianten) im Programm dynamisch geprüft werden und damit Fehler automatisch signalisieren.

"Austesten" ist also weitaus (um viele Größenordnungen) unrealistischer als die sprichwörtliche Suche nach der Stecknadel im Heuhaufen. Testen kann eben nur, wie es Dijkstra formuliert hat, die *Anwesenheit*, aber nicht die *Abwesenheit* von Fehlern zeigen.

Trotzdem ist Testen nach allen Erfahrungen notwendig und nützlich; nur in Fällen, in denen eine Verifikation gelungen ist, könnte man darauf verzichten. Allerdings müßte auch die Verifikation selbst über jeden Zweifel erhaben sein. Wir dürfen aber keinesfalls aus einem erfolglosen, d.h. keine Fehler aufzeigenden Test eine Korrektheitsaussage ableiten.

Der schlimmste Feind des Testers ist offensichtlich die kombinatorische Vielfalt der Parameter, Randbedingungen, globalen Variablen usw. Darum wird der Test um so einfacher (und relativ erfolgreicher) sein, je überschaubarer diese Vielfalt ist. Hier liegt erneut eine Motivation für das Prinzip der hohen Lokalität, das der Bildung von Prozeduren, Modulen, Kapselungen und abstrakten Datentypen zugrundeliegt. Wir testen zweckmäßig bottom-up, also mit kleinsten Einheiten beginnend, die nach dem Test zu größeren Einheiten kombiniert und erneut getestet werden.

4.3.3 Die Konstruktion von Testdaten

Eine *ideale* Testmenge zu einem Programm hat zwei Eigenschaften:

- Wenn das Programm korrekt ist, dann liefert es für *alle* Eingaben aus der Testmenge korrekte Ergebnisse.
- Wenn umgekehrt das Programm für alle Eingaben aus der Testmenge korrekt arbeitet, so ist das Programm auch für alle anderen Eingaben korrekt.

Es gibt nun zwar kein allgemeines Verfahren zur Bestimmung einer idealen Testmenge, jedoch eine Vielzahl von Verhaltensregeln und Rezepten, wie man testet und "halbwegs" ideale Testmengen bestimmt, also Testmengen, die so gewählt sind, daß man, wenn das Programm sie ohne Fehler verarbeitet, mit hoher Wahrscheinlichkeit auf seine Korrektheit schließen kann. Diese Rezepte haben häufig die Form "Man sollte ..." oder "Es empfiehlt sich ..."; keines von ihnen führt jedoch zwingend zum Erfolg. Deshalb bleibt das Testen, die Wahl einer Testmenge und ggf. das Auffinden der fehlerhaften Programmstelle eine sehr *kreative* Tätigkeit, die viel Erfahrung voraussetzt und darüberhinaus ausgesprochen zeitaufwendig ist.

Wie bei der Spurensuche kann man beim Testen entweder völlig unsystematisch vorgehen oder gezielt solche Punkte prüfen, die erfahrungsgemäß oder aufgrund der Programmkonstruktion fehlerträchtig sind. Natürlich verbessert der zweite Ansatz die Chancen auf einen erfolgreichen Test beträchtlich; darum soll er nachfolgend näher betrachtet werden.

Folgende Überlegungen und Beobachtungen liegen der *Auswahl* von Testfällen zugrunde:

- (a) Anforderungen der Spezifikation werden leicht übersehen oder mißverstanden.
- (b) Grenzfälle (z.B. die Verarbeitung einer leeren Liste oder einer Liste maximaler Länge) sind bei der Implementierung oft nicht korrekt behandelt, z.B. weil eine Schleife einmal zu oft oder zu wenig durchlaufen oder weil eine kombinierte Abbruchbedingung in der falschen Reihenfolge getestet wird.
- (c) Fehlersituationen werden bei der Programmkonzeption weniger sorgfältig bedacht als der erwünschte Normalfall.
- (d) Fehler, die in bestimmten Anweisungen oder Verzweigungen stecken, können nur auffallen, wenn diese durchlaufen werden.
- (e) Jede Veränderung des Pfades, auf dem ein Programm durchlaufen wird, kann Fehler offenbaren. Beispielsweise kann eine Initialisierung umlaufen oder eine Inkrementierung fälschlich durchgeführt werden.

Die Verfahren, nach denen man Testfälle auswählt, lassen sich grob in *Black-Box-Test (Schnittstellentest)* und *Glass-Box-Test (programmabhängiger Test)* klassifizieren. Steht das Programm nur als lauffähiges System, nicht durch seinen Programmtext zur Verfügung, so läßt es sich nur an seinen Schnittstellen, d.h. bezüglich der Spezifikation testen. Die Aspekte (d) und (e) aus der obigen Liste lassen sich allerdings nur berücksichtigen, wenn auch der Programmcode zur Verfügung steht. In diesem Fall kann man Testfälle so konstruieren, daß ein möglichst großer Teil der Anweisungen, Verzweigungen oder Pfade durchlaufen werden. Diesen Anteil bezeichnet man als die durch den Test erzielte *Überdeckung (Coverage)*.

Schnittstellentest

Es liegt zunächst nahe, den Schnittstellentest als schlechtere Alternative einzustufen. Tatsächlich hat er aber auch Vorzüge gegenüber dem programmabhängigen Test. Er läßt sich bereits vorbereiten, wenn die Spezifikation vorliegt, und wird nicht durch die Realisierung beeinflusst. Wurde beispielsweise eine Anforderung bei der Realisierung außerachtgelassen, so bietet dieses Verfahren die besseren Chancen; der Glass-Box-Test prüft vor allem, was vorhanden ist, nicht das, was eventuell fehlt. Jeder umfassende Test beginnt darum mit dem Schnittstellentest und wird durch den programmabhängigen Test ergänzt.

Testdaten erzeugt man nach folgendem Prinzip:

- Mindestens eine Prüfung pro spezifizierter Anforderung

Beispiel: Sei die Anforderung, daß eine Fehlermeldung ausgegeben wird, wenn als Name ein leerer String eingegeben wird. Dann muß der leere String unter den Testdaten sein (mit einer Fehlermeldung als Soll-Resultat).

- “Ränder” der Bereiche von beiden Seiten prüfen

Beispiel: Soll eine Telefonnummer zwischen 2 und 18 Ziffern enthalten können, so sind Nummern mit 2 und 18 sowie 1 und 19 Ziffern zu testen. Die beiden Fehlerfälle sollten (im Normalfall, wenn robustes Verhalten des Programms gefordert ist) zu einer akzeptablen Reaktion, nicht zum Absturz führen.

- Normalfälle testen

Neben den Grenzfällen sollte jeweils auch ein “normaler” Fall getestet werden.

Beispiel: Bezogen auf die Situation des letzten Beispiels etwa eine Telefonnummer mit 10 Ziffern.

- Fehlerfälle einzeln testen

Während sich Normalfälle im Test weitgehend kombinieren lassen, müssen Fehlerfälle in der Regel einzeln behandelt werden, da die Erkennung eines Fehlers zum Abbruch führt und weitere Fehler darum nicht mehr erkannt werden können.

Beispiel: In Fortsetzung des bisherigen Beispiels müssen die Fälle "Nummer enthält Buchstaben" und "Nummer ist zu lang" einzeln getestet werden.

- Und niemals die Soll-Resultate vergessen!

Überdeckungskriterien und Überdeckungsgrad

Wie die Überlegungen in 4.3.2 gezeigt haben, ist ein vollständiger Test nicht erreichbar, wenn "vollständig" bedeutet, daß alle möglichen Situationen einzeln getestet werden. Fassen wir aber ähnliche Fälle in einer *Klasse* von Situation zusammen, so haben wir durchaus die Chance, mit dem Test alle Klassen oder einen hohen Anteil von ihnen zu überdecken. Beim Glass-Box-Test orientiert sich die Definition der (Äquivalenz-) Klassen an den Anweisungen, Bedingungen und Verzweigungen (dargestellt z.B. in einem Programmablaufplan) im Programm, nicht an den speziellen Daten. Man kann dieses Verfahren darum auch als *Programmstruktur-abhängigen* Test bezeichnen.

Ein *Programmablaufplan* zu einem Programm ist eine graphische Darstellung, die nach folgenden Regeln aufgebaut ist:

- Jede elementare Anweisung (Zuweisung, Ein-Ausgabeanweisung) und jeder Prozeduraufruf im Programm wird durch einen *Kasten* dargestellt, in den man die zugehörige Anweisung hineinschreibt (Abb. 4.2).

Abb. 4.2 Elementare Anweisung und Prozeduraufruf

- Jede Bedingung einer Schleife oder bedingten Anweisung wird durch eine *Raute* dargestellt, in die die zugehörige Bedingung notiert wird (Abb. 4.3).

Abb. 4.3 Bedingte Anweisung

- Es gibt zwei weitere Symbole *START* und *STOP*, die den Programmanfang bzw. sein Ende repräsentieren (Abb. 4.4).

Abb. 4.4 START- und STOP-Knoten

- Ein Anweisungssymbol A und ein Anweisungs- oder Bedingungssymbol B werden durch einen *Pfeil* verbunden, wenn das Programm nach Ausführen von A zu B übergeht (Abb. 4.5).

Abb. 4.5 Komposition elementarer Anweisungen

- Der mit "wahr" (oder kurz "w") markierte Pfeil einer Raute weist auf das Symbol der Anweisung oder Bedingung, die als erste erreicht wird, wenn die Bedingung zu *TRUE* ausgewertet wird. Der mit "falsch" (oder kurz "f") markierte Pfeil einer

Raute weist auf das Symbol der Anweisung oder Bedingung, die als erste erreicht wird, wenn die Bedingung zu *FALSE* ausgewertet wird.

- Das Darstellungselement für START wird mit der *ersten*, das Darstellungselement für STOP mit jeder als *letzte* auszuführenden Anweisung verbunden.

Man beachte, daß *Sprünge* durch diese Definition ebenfalls erfaßt werden. Sie tauchen in der Darstellung nur als Pfeile auf.

Enthalte ein Programmablaufplan eines zu testenden Programms x Anweisungen (oder Bedingungen). Seien die Testdaten so beschaffen, daß davon y mindestens einmal durchlaufen werden (also $x - y$ nicht durchlaufen werden). Dann erzielen die Testdaten den *Überdeckungsgrad* y/x (Angabe typisch in %). Wir streben natürlich 100 %, also vollständige Überdeckung an. Die Erfahrung zeigt aber, daß dieses scheinbar einfache Ziel in der Praxis kaum erreichbar ist, weil einige Programmteile nur in Sonderfällen zur Ausführung kommen. Daher sind Überdeckungen von 90 % durchaus respektabel (und weit besser als der Durchschnitt).

Ob eine bestimmte Überdeckung erreicht wurde, läßt sich praktisch nur mit Hilfe eines Werkzeugs feststellen, das das Programm *instrumentiert*, also mit zusätzlichen Initialisierungs- und Zählweisungen versieht. Dadurch kann man nach dem Test (oder auch nach einer Serie von Tests) feststellen, welche Anweisungen oder Bedingungen wie oft durchlaufen bzw. ausgewertet wurden.

Wegeüberdeckung

Die Testmenge T eines Programms P heißt *wegeüberdeckend*, wenn es zu jedem Weg W (d.h. jede Folge von Anweisungen und Bedingungen im Programmablaufplan beginnend beim START- und endend beim STOP-Knoten) durch den Programmablaufplan von P mindestens eine Eingabe in T gibt, bei deren Verarbeitung der Weg W durchlaufen wird.

Eine vollständige Wegeüberdeckung setzt voraus, daß die Zahl der Wege endlich und relativ klein ist und sich für jeden Weg Eingabedaten finden lassen. *Endlich* ist sie nicht, wenn das Programm Rekursion oder Schleifen enthält, für die keine obere Grenze angegeben werden kann. *Klein* ist sie nur, wenn diese Grenze niedrig liegt und es nicht zu viele unabhängige Verzweigungen gibt. Die Vorgabe *geeigneter* Eingabedaten kann nahezu unmöglich sein, wenn der Programmcode auch auf komplizierte, nur selten entstehende Situation zugeschnitten ist. Aus diesen Gründen ist die Wegeüberdeckung nur für triviale Programme erreichbar.

Wir hätten also bei vollständiger Wegeüberdeckung einen sehr umfassenden Test. Leider ist diese in der Praxis i.a. unerreichbar, so daß wir die Anforderungen reduzieren müssen. Auf diese Weise kommen wir zur Zweig- und Anweisungsüberdeckung.

Anweisungsüberdeckung

Wird bei Anwendung aller Testdaten ein bestimmter Anteil c der Anweisungen ausgeführt, so haben wir die Anweisungsüberdeckung c (in %) erreicht. Wir können erwarten, bei vollständiger Anweisungsüberdeckung alle Fehler zu erkennen, die darin bestehen, daß Anweisungen eine Wirkung haben, die offensichtlich (d.h. an den Resultaten ablesbar oder sogar durch Absturz sichtbar) sind.

Die Anweisungsüberdeckung ist von allen Überdeckungen die anspruchsloseste, sie ist bei einfachen Algorithmen oft schon mit einem einzigen Testlauf zu erreichen. Allerdings bleiben die logischen Aspekte des Programms weitgehend ungeprüft; insbesondere fällt es nicht auf, wenn Anweisungen unter gewissen Umständen irrtümlich *nicht* ausgeführt werden.

Zweigüberdeckung

Wir definieren darum ein stärkeres Kriterium als die Anweisungsüberdeckung, die Zweigüberdeckung: Eine Testmenge für ein Programm P heißt *zweigüberdeckend*, wenn für jede Bedingung im Programmablaufplan von P jede Verzweigung mindestens einmal durchlaufen wurde. Dazu gehören insbesondere auch die leeren Alternativen von IF-THEN-Anweisungen ohne ELSE.

Bedingungsüberdeckung

Wenn die logischen Bedingungen, von denen die Verzweigung in IF-Anweisungen gesteuert wird, aus mehreren, durch AND oder OR verknüpften Ausdrücken zusammengesetzt sind, ist es zweckmäßig, im Test dafür zu sorgen, daß jeder dieser Teilausdrücke mindestens einmal den Wert TRUE und FALSE hatte. War dies der Fall, so ist Bedingungsüberdeckung erreicht.

Beispiel

An einem kleinen Beispiel wollen wir zeigen, wie unterschiedliche Fehler bei den verschiedenen Überdeckungen auffällig werden. Das Programmbeispiel Summe sei wie folgt spezifiziert:

Summe liefert die Summe der beiden INTEGER-Parameter A und B, soweit diese im Bereich des Typs INTEGER liegt.

Das Beispiel ist so konstruiert, daß die Fehler ins Auge springen. Es wäre natürlich eine Illusion zu glauben, daß die "natürlichen" Fehler immer so auffällig sind.


```

PROCEDURE Summe (A, B: INTEGER): INTEGER; (* Programm P4.8a *)
  VAR S : INTEGER;
  BEGIN (* Summe *)
    S := -A-B;                                     (*1*)
    IF (A # 14127) OR (B # -16999) THEN S := -S; END; (*2*)
    IF (A > 25000) OR (B > 0) THEN INC (S); END;      (*3*)
    IF (A < 25000) OR (B < 0) THEN DEC (S); END;      (*4*)
    IF A = 32323 THEN S := 0; END;                  (*5*)
    RETURN S;                                       (*6*)
  END Summe;

```

Für den Test schreiben wir am besten noch einen *Testrahmen*, der die vollständigen Testdatensätze erhält und Fehler anzeigt.

```

(* Programm P4.8b *)
PROCEDURE Testrahmen (A, B, Soll: INTEGER);
  VAR SummeIst : INTEGER;
  BEGIN (* Testrahmen *)
    WriteInt (A, 6);      WriteString (' = A ');
    WriteInt (B, 8);      WriteString (' = B ');
    WriteInt (Soll, 8);   WriteString (' = Soll');
    SummeIst := Summe (A, B);
    WriteInt (SummeIst, 8); WriteString (' = Summe');
    IF SummeIst # Soll THEN Write (' ***'); END; WriteLn;
  END Testrahmen;

```

Der Schnittstellentest (Black-Box-Test) erfordert in diesem Fall nur relativ wenige Testfälle: Man sollte wenigstens die Kombinationen aus Summanden testen, die positiv, null oder negativ sind, und man sollte Summen bilden, die am unteren oder oberen Rande des Zahlenbereichs (maschinenabhängig!) liegen. Im Beispiel der Prozedur Summe wären dies (bei einem Zahlenbereich von -32 768 bis 32 767) beispielsweise folgende Testdatensätze (d.h. Aufrufe an Testrahmen) :

```

Testrahmen ( 0, 0, 0);
Testrahmen ( 10000, 0, 10000);
Testrahmen ( 0, -30000, -30000);
Testrahmen ( 12345, 4321, 16666);
Testrahmen ( -4321, -12345, -16666);
Testrahmen ( -32000, -768, -32768);
Testrahmen ( 32000, 767, 32767);
Testrahmen ( -32768, 0, -32768);
Testrahmen ( 0, 32767, 32767);
Testrahmen ( 32767, -32767, 0);
Testrahmen ( -32767, 32767, 0);

```

Fehlerfälle sind bei Verzicht auf robustes Verhalten nicht vorzusehen. Die Tests liefern die folgende Ausgabe:

0=A	0=B	0=Sol1	-1=Summe	***
10000=A	0=B	10000=Sol1	9999=Summe	***
0=A	-30000=B	-30000=Sol1	-30001=Summe	***
12345=A	4321=B	16666=Sol1	16666=Summe	
-4321=A	-12345=B	-16666=Sol1	-16667=Summe	***
-32000=A	-768=B	-32768=Sol1	32767=Summe	***
32000=A	767=B	32767=Sol1	-32768=Summe	***
-32768=A	0=B	-32768=Sol1	32767=Summe	***
0=A	32767=B	32767=Sol1	32767=Summe	
32767=A	-32767=B	0=Sol1	0=Summe	
-32767=A	32767=B	0=Sol1	0=Summe	

Es ist bemerkenswert, daß in einigen dieser Tests ein numerischer Überlauf eingetreten ist, ohne daß dieser als Laufzeitfehler sichtbar wurde; er wurde im Resultat durch einen zweiten kompensiert. Das Programm ist in dieser Hinsicht eben systemabhängig.

Wir kommen nun zum programmabhängigen Glass-Box-Test. Vollständige Anweisungsüberdeckung erreichen wir mit einem einzigen Testdatensatz:

32323=A	-1=B	32322=Sol1	0=Summe	***
---------	------	------------	---------	-----

Die Sterne weisen auf die Diskrepanz hin. Eine spätere Analyse wird zeigen, daß die Zeile 5 des Programms verantwortlich ist. Alle übrigen Fehler bleiben vorerst - auch nach einer Beseitigung der Zeile 5 - unerkant.

Um die Zweigüberdeckung zu erreichen, müssen wir dafür sorgen, daß die THEN-Zweige *nicht* durchlaufen werden. Dies leisten zusammen die beiden folgenden Tests:

14127=A	-16999=B	-2872=Sol1	2871=Summe	***
25000=A	0=B	25000=Sol1	25000=Summe	

Das Resultat führt uns auf die Spur des Fehlers in Zeile 2 (der beim Test mit zufällig gewählten Zahlen kaum je auffallen wird).

Die Bedingungsüberdeckung ist durch die vorangegangenen Tests nahezu erreicht. Es fehlt noch die Situation

20000=A	10=B	19990=Sol1	20010=Summe	***
---------	------	------------	-------------	-----

Hier liegt der Fehler offensichtlich in einem falschen Soll-Resultat. Die spätere Wiederholung mit korrektem Soll-Resultat bringt keinen Befund mehr.

Wegeüberdeckung ist in diesem Fall noch relativ leicht erreichbar, denn es kann nicht mehr als $2^4=16$ Pfade geben. Die Analyse zeigt jedoch, daß davon nur sieben tatsächlich möglich sind, die übrigen sind durch die logischen Zusammenhänge (d.h. durch die Werte von A und B) ausgeschlossen. Durch die Tests bisher sind bereits vier verschiedene Fälle erledigt, so daß noch drei zu testen sind:

- nur THEN-Zweig in Anweisungen 2 und 3

26000=A 5000=B 31000=Sol1 31001=Summe ***

- nur THEN-Zweig in Anweisungen 2 und 4

-10000=A -10000=B -20000=Sol1 -20001=Summe ***

- nur THEN-Zweig in Anweisungen 2, 3, 5

32323=A 0=B 32323=Sol1 0=Summe ***

Während die letzte Diskrepanz auf den (bereits vorher entdeckten) Fehler in Zeile 5 zurückzuführen ist, zeigen die beiden anderen Tests einen schwer erkennbaren Fehler an, der nur sichtbar wird, wenn in den Zeilen 3 und 4 nur genau ein THEN-Zweig durchlaufen wird.

Wie man sieht, hat der Glass-Box-Test noch einige Fehler sichtbar werden lassen (Zeilen 2 und 5), die im Black-Box-Test kaum gefunden werden können.

4.3.4 Zusammenfassung

Testen von Programmen trägt, wenn es systematisch geplant und durchgeführt wird, zur Reduktion der Fehler bei; Korrektheit läßt sich durch Testen nicht nachweisen. Ein unsystematischer Test bietet dagegen nicht einmal Gewähr, daß die banalen Fehler aufgedeckt werden.

Darum sollte auch der Test, den der Entwickler selbst vornimmt, gewissen minimalen Anforderungen genügen, vor allem bezüglich Testdatenauswahl einschließlich Vorgabe der Soll-Resultate. Sind die Testbedingungen und Resultate sorgfältig dokumentiert, so kann der Test später mit geringem Aufwand ergänzt und wiederholt werden.

5. Programmierparadigmen und -sprachen

Abschnitt 5.1 zeigt für eine durchgängige Problemstellung (“das Oldenburger Milchkanenproblem”) Entwürfe von Problemlösungen mit fünf verschiedenen *Programmierparadigmen*. Damit soll deutlich werden, daß der imperative Stil von MODULA-2 nur eines unter mehreren Paradigmen für die Programmierung darstellt. Abschnitt 5.2 enthält eine kurze Übersicht über die Entwicklung und Verbreitung von Programmiersprachen.

5.1 Programmierparadigmen

Jeder Programmiersprache liegt ein bestimmtes Konzept zugrunde, ein bestimmtes Begriffssystem, mit dem die Lösung gefaßt (konzipiert) wird. Dieses Konzept nennt man auch ihr *Paradigma*. In Bezug auf solche Paradigmen kann man fünf Kategorien von Programmiersprachen unterscheiden:

- Imperative Sprachen
- Funktionale Sprachen
- Logik-basierte Sprachen
- Objektorientierte Sprachen
- Regelbasierte Sprachen.

Bevor wir anhand einer durchgehenden Problemstellung die unterschiedlichen Paradigmen erläutern, sollen die wesentlichen Charakteristika der einzelnen Kategorien vorgestellt werden.

Imperative Sprachen:

Bei diesen Sprachen besteht ein Programm aus einer Folge von *Befehlen*, wie z.B. “Schreibe in die Variable a den Wert 3”, “Springe an die Stelle x im Programm”. Wesentlich an diesen Sprachen ist das *Variablenkonzept*: Eingabewerte werden in Variablen gespeichert und weiterverarbeitet. In diesen Sprachen spiegelt sich die Architektur des *Von-Neumann-Rechners* wieder.

Beispiele für imperative Sprachen sind Cobol, PASCAL und MODULA-2.

Funktionale Sprachen:

Programme werden als *mathematische Funktionen* betrachtet. Die Eingabewerte dieser Funktionen werden als Parameter bezeichnet.

In *rein funktionalen* Programmiersprachen werden die Funktionen durch Zusammensetzung aus einfacheren Funktionen definiert, wobei bestimmte Grundfunktionen vorgegeben sind.

In *applikativen* Programmiersprachen wird dagegen eine Funktion durch Anwendung von Funktionen auf ihre Parameter definiert, wobei als Parameter wiederum Funktionen auftreten können. Programme sind ein System von Funktionsdefinitionen.

Die bekannteste funktionale Programmiersprache ist LISP, eine neuere Entwicklung ist Miranda.

Logik-basierte Sprachen:

Bei diesen Sprachen wird Programmierung als *Beweisen* in einem System von Tatsachen und Schlußfolgerungen aufgefaßt: Man gibt eine Menge von *Fakten* (gültige Prädikate), *Regeln* (Aussagen, wie man aus Fakten neue Fakten gewinnt) und eine *Abfrage* vor, die bei der Ausführung mit JA oder NEIN beantwortet wird, ggf. auch mit den für JA infragekommenden Belegungen der Variablen in der Abfrage.

Die bekannteste Logik-basierte Sprache ist PROLOG.

Objektorientierte Sprachen:

Bei diesen Sprachen werden die zum Lösen eines Problems notwendigen Informationen (Daten und Anweisungen bzw. Regeln) in *Objekten* zusammengefaßt. Objekte können durch Senden von *Nachrichten* (Mitteilungen an andere Objekte) miteinander Informationen austauschen. Jedes Objekt gehört zu einer bestimmten *Klasse*, durch die festgelegt ist, welche Nachrichten es versteht und wie es auf diese reagiert.

Beispiele objektorientierter Sprachen: Smalltalk, Eiffel und C++.

Regelbasierte Sprachen:

Das grundlegende Konzept dieser Sprachen sind "*wenn-dann-Regeln*" etwa der Form "*wenn* <Bedingung> *gilt, dann* <Aktion> *ausführen*". Die einer Problemstellung zugrundeliegende Welt wird in Objekte zerlegt, deren Zustandsübergänge in Form von Regeln beschrieben werden.

Beispiel einer regelbasierten Sprache: OPS5.

Problemstellung

Es handelt sich um ein alltägliches Problem, wie es einem in Oldenburg und Umgebung begegnet:

"Man kommt in einen Milchladen mit einem Gefäß, mit dem man genau 7 Liter abmessen kann, und möchte 4 Liter Milch kaufen. Der Milchverkäufer besitzt leider nur einen Eimer mit der Eichmarke bei genau 10 Litern. Frage: Kann man mit diesen beiden Gefäßen aus einer riesigen Milchwanne genau 4 Liter abfüllen oder nicht?"

Die Lösung sollte von der konkreten Problemstellung abstrahieren und allgemein, also für beliebige Mengen anwendbar sein.

Nachfolgend werden für die aufgezählten fünf Programmierparadigmen die jeweils typischen Konzepte am Beispiel des "Oldenburger Milchkanenproblems" vorgestellt. Dabei muß berücksichtigt werden, daß nicht jedes Problem für jedes Paradigma geeignet ist, so daß der Vergleich anhand eines konkreten Problems unvermeidlich auch falsches suggeriert.

5.2.1 Imperatives Programmieren

Man überläßt die Arbeit des Schöpfens, Umfüllens und Ausgießens von Milch einem "Knecht", der *Befehle* genauestens ausführt, aber gänzlich ohne eigene Intelligenz ist. In diesem Fall muß man die Lösung *selbst* entwickeln und in einer Befehlsfolge niederlegen, die der Knecht anschließend präzise *abarbeitet* (G bezeichnet dabei das 7-Liter-Gefäß, E den 10-Liter-Eimer):

Handlung (für 4 Liter)	danach sind	y Liter in G und	x Liter in E
Fülle G	y = 7	x = 0	
Umfüllen von G nach E		y = 0	x = 7
Fülle G		y = 7	x = 7
Umfüllen von G nach E		y = 4	x = 10
Ausgießen E (fertig)		y = 4	x = 0

Wollte man dagegen 5 Liter kaufen, so muß man einen völlig neuen Lösungsweg in Form einer Folge von folgenden 18 Befehlen erarbeiten:

Fülle E; Umfüllen E nach G; Ausgießen G; Umfüllen E nach G;
 Fülle E; Umfüllen E nach G; Ausgießen G; Umfüllen E nach G;
 Fülle E; Umfüllen E nach G; Ausgießen G; Umfüllen E nach G;
 Ausgießen G; Umfüllen E nach G;
 Fülle E; Umfüllen E nach G; Ausgießen G; Umfüllen E nach G.

Dies liefert tatsächlich das gewünschte Ergebnis "5 Liter im Gefäß G". Man erkennt, daß gewisse Handlungen sich wiederholen. Im Falle, daß 5 Liter abzufüllen sind, entdeckt man 4 Wiederholungen mit einer kleinen Abweichung.

In imperativen Sprachen, z.B. in MODULA-2, gibt es hierfür Abkürzungen:

```
FOR j := 1 TO 4 DO
  Fuelle E; Umfuellen E nach G; Ausgiessen G; Umfuellen E nach
  G;
  IF j = 3 THEN Ausgiessen G; Umfuellen E nach G; END;
END;
```


Die Abweichung für $j = 3$ kommt dadurch zustande, daß sich im Eimer E noch mehr befindet, als in G maximal hineinpaßt. Statt der Laufschleife (for $j := 1$ to 4) verwendet man dann besser eine "solange"-Wiederholung: "Solange in E noch mindestens soviel Milch ist, wie maximal in G paßt, tue ...".

Die Übertragung in imperativen Pseudocode liefert "WHILE ... DO ...", und so erhält man eine "WHILE $E \geq 7$ DO ... -Schleife". Als Abbruchbedingung der Iteration wählt man "in G sind genau 5 Liter", bzw. man wiederholt die Iteration solange, wie nicht 5 Liter in G sind. In Pseudocode: "WHILE $G \neq 5$ DO ... END;".

```
(* Anfangs sind G und E leer *)
WHILE G # 5 DO
  Fuelle E;
  WHILE E >= 7 DO
    Umfuellen von E nach G;   Ausgiessen G;
  END (* WHILE *);
  Umfuellen von E nach G;
END (* WHILE *);
```

Hier wurden die Behälter G und E bereits als "Variablen" aufgefaßt, deren Inhalte man kennt und vergleichen kann. Die einzelnen Operationen lauten in einer Programmiersprache:

"Fülle E"	entspricht $E := 10$
"Ausgießen G"	entspricht $G := 0$
"Umfüllen von E nach G"	entspricht falls $E + G \leq 7$ dann $G := G + E$; $E := 0$ (anschließend ist E leer) andernfalls $E := E - (7 - G)$; $G := 7$ (anschließend ist G voll)

Die obige Lösung in MODULA-2 lautet dann:

```
PROCEDURE Fuenfliter;                                (* Programm P5.1 *)
VAR G, E : INTEGER;
BEGIN (* FuenfLiter *)
  G := 0;   E := 0;
  WHILE G # 5 do
    E := 10;
    WHILE E >= 7 DO
      E := E - (7 - G);   G := 7;   G := 0;
    END (* WHILE E...*);
    G := G + E;   E := 0
  END (* WHILE G...*);
END (* FuenfLiter *);
```

Als erstes muß man im obigen Programmstück die Handlungsanweisungen für den "Knecht" ausgeben. Dies geschieht mit write- oder writeln-Befehlen. Als zweites teilt man ständig mit, die wievielte Handlung gerade auszuführen ist, damit der Knecht genau informiert ist.

Zur Verallgemeinerung ersetzen wir die Zahlen 7 und 10 durch variable Größen Gmax und Emax. Dann funktioniert das gleiche Programm immer noch, aber nicht mehr für alle Werte: Wenn d der größte gemeinsame Teiler von Gmax und Emax ist, dann kann man nur solche Mengen abfüllen, die ein Vielfaches von d sind. Wir berechnen im folgenden Programm daher den größten gemeinsamen Teiler (ggT) und brechen ab, falls der ggT die gewünschte Literzahl der Sollmenge nicht teilt.

```

MODULE OMP;                                     (* Programm P5.2 *)
(* Oldenburger Milchkannenproblem *)
FROM InOut IMPORT WriteString, WriteInt, ReadInt, WriteLn;
VAR ZeilenNr   : INTEGER;
    Gmax, Emax : INTEGER;
    Sollmenge  : INTEGER;

PROCEDURE schreib (Zeile : ARRAY OF CHAR);
BEGIN (* schreib *)
    INC (ZeilenNr);  WriteInt (ZeilenNr, 0);
    WriteString ('-ter Schritt: ');
    WriteString (Zeile);  WriteLn;
END schreib;

PROCEDURE DruckeAnweisung (Emax, Gmax, Sollmenge : INTEGER);
(* Druckt eine Anweisungssequenz, um im Gefaess die Soll-
   Menge bereitzustellen. *)
(* Emax      ist das Fassungsvermoegen des Eimers      *)
(* Gmax      ist das Fassungsvermoegen des Gefaesses *)
(* 0 < Sollmenge <= Gmax; alle Angaben stehen in Liter *)
VAR  G, E: INTEGER; (* Inhalt Gefaess und Eimer *)

PROCEDURE LeereG;
BEGIN schreib ('Ausgiessen G.');
```

G := 0; END LeereG;

```

PROCEDURE FuelleE;
BEGIN schreib ('Fuelle E.');
```

E := Emax; END FuelleE;

```

PROCEDURE EnachG;
VAR GesMenge : INTEGER;
BEGIN (* EnachG *)
    schreib ('Eimer in Gefaess umfuellen');
```

WHILE (E > 0) AND (G < Gmax) DO DEC (E); INC (G); END;

END EnachG;

```

PROCEDURE GGT (A, B : INTEGER) : INTEGER;
(* iterative Loesung *)
VAR Rest: INTEGER;
BEGIN (* GGT *)
    WHILE A > 0 DO  Rest := B MOD A; B := A; A := Rest;  END;
```

RETURN B;

END GGT;

```

BEGIN (* DruckeAnweisung *)
    ZeilenNr:= 0;
    IF Sollmenge MOD GGT (Gmax, Emax) = 0 THEN
```

```

(* Loesung moegl. *)
G := 0;   E := 0; (* Anfangszustand *)
WHILE G # Sollmenge DO
  FuelleE;   EnachG;
  REPEAT   LeereG;   EnachG;   UNTIL E = 0;
END (* WHILE *);
WriteString ('Fuer ein Gefaess mit ');
WriteInt (Gmax,0);
WriteString (' Litern und einem Eimer mit ');
WriteInt (Emax,0);
WriteString (' Litern erhaelt man das Resultat nach ');
WriteInt (Zeilenr,0);   WriteString (' Schritten. ');
WriteLn;
WriteString ('Dann enthaelt G ');
WriteInt (Sollmenge,0);
WriteString (' Liter und E ist leer. ');   WriteLn;
ELSE
  WriteString ('Loesung nicht moeglich. ');   WriteLn;
END (* IF *);
END DruckeAnweisung;

BEGIN (* OMP *)
  WriteLn;
  REPEAT
    WriteString ('Gmax, Emax, Sollmenge eingeben: ');
    ReadInt (Gmax);   ReadInt (Emax);   ReadInt (Sollmenge);
    UNTIL (0 <= Sollmenge) AND (Sollmenge <= Gmax) AND
          (0 < Gmax) AND (Gmax <= Emax);
    DruckeAnweisung (Emax, Gmax, Sollmenge);
  END OMP.

```

Dieses Programm liefert z.B. für ein Gefäß mit 7 Litern und einen 10-Liter-Eimer für eine Füllung von 5 Litern (Sollmenge = 5 eingeben) folgende Ausgabe (die Zahlen 7, 10 und 5 stellen die Eingabe dar):

Gmax, Emax, Sollmenge eingeben:

7 10 5

- 1-ter Schritt: Fuelle E.
- 2-ter Schritt: Umfuellen von E nach G.
- 3-ter Schritt: Ausgiessen G.
- 4-ter Schritt: Umfuellen von E nach G.
- 5-ter Schritt: Fuelle E.
- 6-ter Schritt: Umfuellen von E nach G.
- 7-ter Schritt: Ausgiessen G.
- 8-ter Schritt: Umfuellen von E nach G.
- 9-ter Schritt: Fuelle E.
- 10-ter Schritt: Umfuellen von E nach G.
- 11-ter Schritt: Ausgiessen G.
- 12-ter Schritt: Umfuellen von E nach G.
- 13-ter Schritt: Ausgiessen G.

14-ter Schritt: Umfüllen von E nach G.

15-ter Schritt: Fülle E.

16-ter Schritt: Umfüllen von E nach G.

17-ter Schritt: Ausgießen G.

18-ter Schritt: Umfüllen von E nach G.

Für ein Gefäß G mit 7 Litern und einen Eimer E mit 10 Litern erhält man das Resultat nach 18 Schritten. Dann enthält G 5 Liter und E ist leer.

Man kann nun kompliziertere Beispiele nehmen, etwa einen 40-Liter-Eimer, mit dessen Hilfe man in ein 31 Liter-Gefäß genau 27 Liter füllen soll:

Gmax, Emax, Sollmenge eingeben:

31 40 27

1-ter Schritt: Fülle E.

2-ter Schritt: Umfüllen von E nach G.

3-ter Schritt: Ausgießen G.

4-ter Schritt: Umfüllen von E nach G.

5-ter Schritt: Fülle E.

6-ter Schritt: Umfüllen von E nach G.

7-ter Schritt: Ausgießen G.

8-ter Schritt: Umfüllen von E nach G.

9-ter Schritt: Fülle E.

10-ter Schritt: Umfüllen von E nach G.

11-ter Schritt: Ausgießen G.

12-ter Schritt: Umfüllen von E nach G.

Für ein Gefäß G mit 31 Litern und einen Eimer E mit 40 Litern erhält man das Resultat nach 12 Schritten. Dann enthält G 27 Liter und E ist leer.

Wählt man hier 4 statt 27 Liter, so benötigt man 128 Schritte.

5.2.2 Funktionales Programmieren

Beim funktionalen Programmieren überlegt man sich, welche *Grundfunktionen* durchgeführt werden müssen und wie diese miteinander *zusammenhängen*. Denkt man hier in sehr einfachen Grundfunktionen, so nähert sich die Denkweise der der imperativen Programmierung.

Wir beschreiben die Funktion, die der Knecht ausführt, wenn er in einer beliebigen *Situation* ist: Im Eimer E befinden sich x Liter, im Gefäß sind y Liter, und als sonstige Information muß man noch wissen, daß E genau $E_{\max}$ und G genau $G_{\max}$ Liter fassen, und daß man am Ende Sollmenge Liter im Gefäß G haben soll. Die Funktion besitzt also die fünf Angaben ("*Parameter*") x , y , $E_{\max}$, $G_{\max}$, Sollmenge. Zunächst gehen wir davon aus, daß in der Situation kein Fehler vorliegt (z.B. sind die

Literzahlen nicht etwa negative Zahlen, oder im Gefäß sind mehr Liter, als $G_{\max}$ angibt, usw.). Dann erhält man folgende *Funktion* S (entspricht der Angabe, wie man aus der *gegenwärtigen* Situation eine *neue* Situation schafft):

$S(x, y, E_{\max}, G_{\max}, \text{Sollmenge}) =$

falls $y = \text{Sollmenge}$	dann ist man fertig (d.h. NIL = nichts tun) sonst:
falls $y = G_{\max}$	dann "Ausgießen G", d.h. $S(x, 0, E_{\max}, G_{\max}, \text{Sollmenge})$, sonst:
falls $x = 0$	dann "Fülle E", d.h. $S(E_{\max}, y, E_{\max}, G_{\max}, \text{Sollmenge})$ sonst:
falls $x + y \geq G_{\max}$	dann "Umfüllen E nach G", wobei ein Rest in E bleibt, d.h. $S(x - G_{\max} + y, G_{\max}, E_{\max}, G_{\max}, \text{Sollmenge})$, sonst:
falls $x + y < G_{\max}$	dann "Umfüllen E nach G", wobei E leer wird, d.h. $S(0, x + y, E_{\max}, G_{\max}, \text{Sollmenge})$.

In Pseudocode kann man die Funktion S folgendermaßen notieren (hierbei lassen wir die letzte Bedingung " $x + y < G_{\max}$ " weg, da sie automatisch erfüllt sein muß):

```
define function S (x, y, Emax, Gmax, Sollmenge) =      (* Programm P5.3a *)
  if y = Sollmenge then NIL
  else if y = Gmax then concat ('Ausgießen G', S (x, 0, Emax, Gmax, Sollmenge))
  else if x = 0 then concat ('Fülle E', S (Emax, y, Emax, Gmax, Sollmenge))
  else if x + y >= Gmax then concat ('Umfüllen von E nach G',
                                     S (x - Gmax + y, Gmax, Emax, Gmax, Sollmenge))
  else concat ('Umfüllen von E nach G',
               S (0, x + y, Emax, Gmax, Sollmenge)).
```

Die Funktion S wird definiert durch Anwendung der Funktionen "if-then-else", "=", ">=", "+", "-" und "concat" auf die Parameter. Man beachte, daß diese Funktion *rekursiv* definiert ist. Dies ist eine oder sogar *die* zentrale Methode der funktionalen Programmierung. Die Funktion S berechnet mit Hilfe der Funktion concat eine Liste von Anweisungen, die genau der Handlungsfolge für den oben genannten Knecht entspricht. Wir müssen nur noch die Anfangssituation mit Hilfe einer zweiten Funktion A festlegen:

```
define function A (Emax, Gmax, Sollmenge) =      (* Programm P5.3b *)
  if Gmax < 1 then concat ('Gefäß nicht vorhanden.', NIL)
  else if Sollmenge > Gmax then concat (('Gefäß zu klein.'), NIL)
  else if Sollmenge < 0 then concat (('Negative Literzahl nicht möglich.'), NIL)
  else if Gmax > Emax then concat (('Rollentausch Eimer / Gefäß.'),
                                   A (Gmax, Emax, Sollmenge))
  else if Sollmenge mod ggT (Emax, Gmax) <> 0 then
    concat ('Keine Lösung möglich.', NIL)
  else S (0, 0, Emax, Gmax, Sollmenge).
```

“Sollmenge mod ggT (E_{max}, G_{max}) $\diamond 0$ ” bedeutet: Die gewünschte Literzahl Sollmenge läßt sich nicht durch den ggT von E_{max} und G_{max} teilen. In diesem Fall ist keine Lösung möglich, weil beim Umschütten, Füllen und Ausgießen immer ein Vielfaches des ggT in den Behältern sein muß. Der ggT lautet für nicht-negative Zahlen ($a \bmod b$ ist der Rest, der bei Division von a durch b bleibt):

```
define function ggT (a,b) = if b = 0 then a else ggT (b, a mod b).
```

Ein Programm in einer funktionalen Programmiersprache besteht aus einer Folge von *Funktionsdefinitionen* und einem konkreten *Funktionsaufruf*, in dem die Anfangswerte angegeben werden. In unserem Beispiel also:

```
define function S (x, y, Emax, Gmax, Sollmenge) =      (* Programm P5.3c *)
    .... wie oben angegeben ....
define function A (Emax, Gmax, Sollmenge) =
    .... wie oben angegeben ....
define function ggT (a, b) = if b = 0 then a else ggT (b, a mod b).
A (7, 10, 5)
```

Der Funktionsaufruf A (7, 10, 5) bewirkt nun, daß die Funktion A für diese aktuellen Werte *ausgerechnet* wird, wobei von der Funktion A auf die Funktionen ggT und S zurückgegriffen wird; die entsprechende Auswertung von S liefert hierbei die Handlungsfolge als Ausgabe. Man erhält in diesem Fall (7, 10, 5) genau die oben bereits aufgelisteten 18 Handlungen. Analog kann man A (31, 40, 27) ausrechnen lassen usw.

5.2.3 Logik-basiertes Programmieren

Bei der prädikativen oder Logik-basierten Programmierung versucht man, das Problem durch seine *Eigenschaften* (“Prädikate”) in Form von *Fakten* und *Regeln* zu erfassen. Was sind nun die Eigenschaften beim Milchkannen-Problem?

Da ist zunächst die “Ziel”-Eigenschaft: *Wenn* Sollmenge Liter im Gefäß G sind, *dann* ist das Ziel erreicht. Wir notieren also:

ziel (y, h) $\leftarrow$ situation (x, y, h), gültig (x, y),

wobei links von “ $\leftarrow$ ” die *Implikation* und rechts davon die durch Kommata getrennten, als *konjunktiv* verknüpft verstandenen *Prämissen* stehen.

Die Variable h steht dabei für die zu berechnende Handlungsfolge für den Knecht. situation (x, y) soll bedeuten: Der Eimer E und das Gefäß G erfüllen die Eigenschaft, daß genau x Liter in E und y Liter in G sind. Wir wollen nur *gültige*, also *tatsächlich mögliche* Situationen betrachten. Hierfür verwenden wir die Eigenschaft gültig:

gültig (x, y) $\leftarrow$ (x $\geq$ 0), (y $\geq$ x), (x $\leq$ E_{max}), (y $\leq$ G_{max})

Dies ist so zu lesen: Wenn x größer oder gleich null ist und wenn y größer oder gleich x ist und wenn x kleiner oder gleich $E_{\max}$ ist und wenn y kleiner oder gleich $G_{\max}$ ist, dann liegt eine gültige Situation vor. Die erste Zeile bedeutet: Ein Ziel für den Wert y liegt vor, wenn man eine gültige Situation mit x Litern in E und y Litern in G hat. Dabei spielt die Größe von x hierbei keine Rolle, deshalb tritt sie auf der linken Seite nicht auf. Weiterhin muß die vorhandene Situation gültig gewesen sein.

Nun geben wir an, wie Operationen auf Situationen arbeiten.

1. situation ($x, y, [h, \text{"Fülle E."}]$)	←	situation ($0, y, h$), ($x = E_{\max}$)
2. situation ($x, y, [h, \text{"Fülle G."}]$)	←	situation ($x, 0, h$), ($y = G_{\max}$)
3. situation ($x, y, [h, \text{"Ausgießen E."}]$)	←	situation (u, y, h), ($x = 0$)
4. situation ($x, y, [h, \text{"Ausgießen G."}]$)	←	situation (x, v, h), ($y = 0$)
5. situation ($x, y, [h, \text{"Von G nach E."}]$)	←	situation (u, v, h), ($x = u + v$),
($x < E_{\max}$), ($y = 0$)
6. situation ($x, y, [h, \text{"Von E nach G."}]$)	←	situation (u, v, h), ($y = u + v$),
($y < G_{\max}$), ($x = 0$)
7. situation ($x, y, [h, \text{"Von G nach E."}]$)	←	situation (u, v, h), ($x = E_{\max}$),
($x + y = u + v$)
8. situation ($x, y, [h, \text{"Von E nach G."}]$)	←	situation (u, v, h), ($y = G_{\max}$),
($x + y = u + v$)

Die in den Regeln vorkommenden Variablen haben nur lokale Gültigkeit, d.h. sie gelten nur für jeweils eine Regel. Außerdem werden sie bei jeder Anwendung einer Regel neu erzeugt.

Jetzt muß man noch wissen, welche Situationen ohne alle Voraussetzungen herbeigeführt werden können. Dieses Wissen faßt man in sogenannten *Fakten* zusammen. Dies sind logische Regeln, die stets gelten, die also keiner Prämisse (Voraussetzung) auf der rechten Seite von " $\leftarrow$ " bedürfen. Hier genügt eine Situation:

situation ($0, 0, []$) ←

Das Programm besteht nun aus der *Abfrage*, ob eine Eigenschaft gilt. Hier fügt man die Fakten

($E_{\max} = 10$) ←

($G_{\max} = 7$) ←

hinzu und fragt dann

ziel ($3, h$) ?

Ein *Logik-Interpreter* (auch *Beweiser* genannt) versucht nun, die gewünschte Eigenschaft aus Fakten und Regeln herzuleiten. Dabei werden die Variablen schrittweise mit Werten belegt, so daß die Fakten und Regeln richtig bleiben.

Zum Beispiel bewirkt $\text{ziel}(3, h)$, daß die Variable y in obiger Regel für ziel nur den Wert 3 annehmen darf und die Variable h nach Herleitung des Ziels die gesuchte Anweisungsliste als Wert besitzt. Man erhält also aus

$$\text{ziel}(y, h) \leftarrow \text{situation}(x, y, h), \text{gültig}(x, y)$$

die Information: $y = 3$.

Man prüft, ob man ein x so finden kann, daß die Eigenschaften $\text{situation}(x, 3, h)$ und $\text{gültig}(x, 3)$ zutreffen. Zum Beispiel kann man aus der 1. Regel

$$\text{situation}(x, y, [h, \text{"Fülle E."}]) \leftarrow \text{situation}(0, y, h), (x = \text{Emax})$$

zusammen mit dem Fakt " $(\text{Emax} = 10)$ " logisch die *Folgerung*

$$\text{situation}(10, 3, [h, \text{"Fülle E."}]) \leftarrow \text{situation}(0, 3, h) \text{ schließen, d.h.:}$$

Man kann das Ziel $\text{ziel}(3, [h, \text{"Fülle E."}])$ erreichen, wenn man zuvor die Situation $\text{situation}(0, 3, h)$ herstellen konnte.

Mit der 6. Regel

$$\text{situation}(x, y, [h, \text{"Von G nach E."}]) \leftarrow \text{situation}(u, v, h), (y = u + v), \\ (y < \text{Gmax}), (x = 0)$$

kann man nun folgern, daß $\text{situation}(0, 3, [h, \text{"Von G nach E."}])$ herstellbar ist, wenn auch $\text{situation}(3, 0, h)$ herstellbar ist. Eine weitere Inspektion der Regeln zeigt, daß man folgende *Ableitungskette* (incl. Nummer der angewendeten Regel) verwenden kann:

$\text{situation}(0, 0, [])$	
$\text{situation}(10, 0, [\text{"Fülle G."}])$	1. Regel
$\text{situation}(3, 7, [\text{"Fülle G."}, \text{"Von E nach G."}])$	8. Regel
$\text{situation}(3, 0, [\text{"Fülle G."}, \text{"Von E nach G."}, \text{"Ausgießen G."}])$	4. Regel
$\text{situation}(0, 3, [\text{"Fülle G."}, \text{"Von E nach G."}, \\ \text{"Ausgießen G."}, \text{"Von E nach G."}])$	6. Regel
$\text{gültig}(0, 3)$	
$\text{ziel}(3, [\text{"Fülle G."}, \text{"Von E nach G."}, \text{"Ausgießen G."}, \text{"Von E nach G."}])$	

Die Ausgabevariable h besitzt dann als Wert eine Liste von vier Handlungen: Fülle E, Umfüllen von E nach G, Ausgießen G, Umfüllen von E nach G. Das Prädikat *gültig* ist hier eigentlich überflüssig, da die Regeln so formuliert wurden, daß die entstehenden Situationen stets *gültig* sind.

Das Problem in der Praxis liegt darin, daß man einen Logik-Interpreter entwickeln und implementieren muß, der *alle* Möglichkeiten *durchtestet*. Hierfür hat man bei der Sprache PROLOG spezielle Techniken erarbeitet, die jedoch auf die Sprache zurückwirken. Für die Effizienz spielt es z.B. eine große Rolle, in welcher Reihenfolge die einzelnen Fakten und Regeln durchmustert werden und wie man die Regeln abarbeitet. Wenn man dies nicht weiß, so kann man den Interpreter leicht in

unendliche Schleifen “locken”, die man durch kleine Änderungen des Programms hätte verhindern können.

5.2.4 Objektorientiertes Programmieren

Beim objektorientierten Programmieren versucht man *Objekte* zu identifizieren, die in der Aufgabenstellung auftauchen. Für das aktuelle Problem sind dies:

- der Eimer (mit Fassungsvermögen $E_{\max}$)
- das Gefäß (mit Fassungsvermögen $G_{\max}$)
- die Milchwanne (mit beliebig viel Milch)
- der Knecht.

Wir beschreiben diese Objekte jetzt “aus ihrer jeweiligen Sicht” genauer und versuchen gleichzeitig, Objekte zu *Objektklassen* zu *abstrahieren*. Zunächst kann man den Eimer und das Gefäß als gleichartig auffassen, d.h. als Spezialfälle einer Objektklasse Behälter. Diese Objektklasse läßt sich wie folgt bezüglich ihrer Eigenschaften (nachfolgende Punkte 1 und 2) sowie der auf ihr ausführbaren Operationen (Punkte 3 - 5), *Methoden* genannt, beschreiben:

1. Maximales *Fassungsvermögen* MAX (eine natürliche Zahl)
2. Aktueller *Inhalt* INH (eine natürliche Zahl zwischen 0 und dem maximalen Fassungsvermögen)
3. Kann *aufgefüllt* werden auf dieses maximale Fassungsvermögen, indem eine entsprechende Anforderung an die Milchwanne gestellt wird
4. Kann *entleert* werden, indem der Inhalt in die Milchwanne geschüttet wird
5. Kann von einem anderen Behälter Milch bis zur Differenz aus dem maximalen Fassungsvermögen und dem aktuellen Inhalt *zugeschüttet bekommen*.

Schreiben wir dies etwas formaler auf:

Definition Objektklasse Behälter:

Variablen MAX: integer, ($MAX > 0$)

INH: integer, ($0 \leq INH$), ($INH \leq MAX$), anfangs ist $INH = 0$;

Methoden

Füllen: Milchwanne um $(MAX - INH)$ Liter bitten;

Leeren: An Milchwanne INH Liter schicken;

Zuschütten aus anderem Behälter:

Umschütten von $(MAX - INH)$ Liter aus Behälter A

Abgeben von X Liter an anderen Behälter:

falls $X < INH$ ist,

dann gib X Liter ab, vermindere INH um X

sonst gib INH Liter ab, setze $INH = 0$;

Ende der Definition von Objektklasse Behälter.

Mit Hilfe der bisher eingeführten programmiersprachlichen Elemente kann man dies weiter präzisieren:

```

Define Objectclass Behälter:                                     (* Programm P5.4a *)
  Var  MAX, INH : integer,
      A : Bezeichner für anderen Behälter,
      with (MAX > 0), (0 <= INH), (INH <= MAX),
      initial : INH := 0;
  Methods
    Füllen: begin Milchwanne.Hole (MAX-INH); INH := MAX end;
    Leeren: begin Milchwanne.Leeren (INH); INH := 0 end;
    Zuschütten:
      begin INH := INH + A.Übernehmen (MAX-INH) end;
    Abgeben (X : integer) : integer;
      begin
        if X < INH then INH := INH-X else X:= INH; INH := 0;
        return X
      end;
end Objectclass Behälter;

```

Wenn beim Zugriff auf eine Variable oder Aufruf einer Methode keine genauere Behälterangabe "A. ..." dabei steht, dann ist stets der Behälter selbst gemeint. Man kann sich nun zwei Objekte der Klasse Behälter durch den Befehl *new* verschaffen ("kreieren"). Man muß den Behältern Namen geben und einige oder alle der Variablen mit Werten füllen oder Bezeichner mit Objekten identifizieren:

```

Eimer := new Behälter (MAX = 10, A = Gefäß);
Gefäß := new Behälter (MAX = 7, A = Eimer);

```

Die Milchwanne führen wir nun als Objekt einer neuen Klasse Wanne ein, das aber keine Aktivitäten durchführen muß, da diese bereits im Objekt Behälter vorgesehen sind.

```

Define Objectclass Wanne:                                       (* Programm P5.4b *)
  Methods
    Hole (i : integer) : NIL;
    Leeren (i : integer) : NIL;
end Objectclass Wanne;

Milchwanne := new Wanne;

```

Der Knecht muß nun die Reihenfolge, in der die Objekte tätig werden sollen, überwachen, d.h. wir werden Knecht als Objekt einer allgemeinen Klasse von Steuerobjekten einführen. Konkret muß sich der Knecht den Wert Sollmenge der gewünschten Literzahl merken:


```

Define Objectclass Steuerung:                                     (* Programm P5.4c *)
  Var  Sollmenge: integer;
       Bgroß, Bklein: Bezeichnungen für Behälter;
       with (0 <= Sollmenge), (Sollmenge <= Bklein.MAX);
  Methods
    Umfüllen:
      if Sollmenge = Bklein.MAX then Bklein.Füllen
      else
        while Bklein.INH <> Sollmenge do
          if Bklein.INH = Bklein.MAX then Bklein.Leeren
          else if Bgroß.INH = 0 then Bgroß.Füllen
          else Bklein.Zuschütten;
end Objectclass Steuerung;

Knecht := new Steuerung (Sollmenge = 5, Bgroß = Eimer, Bklein = Gefäß);

Knecht.Umfüllen;

```

Die letzte Zeile bewirkt das Ausführen der Handlungen: Das Objekt Knecht wird aufgefordert, seine Methode (d.h. den ihm bekannten Algorithmus) Umfüllen auszuführen.

Diese Methode prüft zunächst, ob die dem Knecht mitgeteilte Zahl 5 (für Sollmenge) gleich dem Wert Gefäß.MAX, also 7 ist. Dies trifft nicht zu, daher wird nun die while-Schleife ausgeführt. Da anfangs Eimer.INH = 0 ist, wird als erstes Eimer.Füllen aufgerufen. Anschließend besitzt die Variable INH des Behälters Eimer den Wert 10 (entspricht Eimer.MAX).

Nun wird die while-Schleife erneut durchlaufen. Diesmal wird Gefäß.Zuschütten ausgeführt. Gefäß fordert dann Eimer.Abgeben an. Diese Methode bewirkt, daß Gefäß.INH den Wert 7 und Eimer.INH den Wert $10 - (7 - 0) = 3$ bekommen.

Der nächste Schleifendurchlauf ruft Gefäß.Leeren auf usw., bis schließlich die Zahl Sollmenge erreicht wird. Trifft dies nie zu, dann wird der Knecht ewig beschäftigt sein (entspricht nicht lösbarer Aufgabe). Dieser Fall wird in der in Programm P5.4c vorgestellten Lösung jedoch nicht abgefangen.

5.2.5 Regelbasiertes Programmieren

Beim regelbasierten Programmieren beschreibt man die "Welt", die durch die Aufgabenstellung definiert ist, durch *Datenbereiche*, und knüpft Handlungen auf diesen Datenbereichen an *Regeln* der Form

WENN eine Bedingung zutrifft DANN führe folgende Aktion aus.

Die Welt kann in unserem Beispiel durch einen Zettel beschrieben werden, auf dem die Werte von Emax und Gmax stehen und auf dem notiert ist, wieviel Liter sich gerade in E und in G befinden. Zusätzlich schreibt man hierauf alle noch nicht abgearbeiteten *Anforderungen* ("Ziele") auf. Eine *Anfangssituation* lautet z.B.

Weltbeschreibung: Emax = 10, Gmax = 7, G = 0, E = 0, ziel (G = 3).

Durch *Wenn-Dann-Regeln* (im folgenden nur Regeln genannt) gibt man an, wie diese Weltbeschreibung sich ändern darf; hier genügen folgende neun Regeln:

- (1) WENN G = 0 und ziel (G = z)
DANN merke ("G = Gmax"), streiche ("G = 0"),
protokolliere ("Fülle G.").
- (2) WENN E = 0 und ziel (G = z)
DANN merke ("E = Emax"), streiche ("E = 0"),
protokolliere ("Fülle E.").
- (3) WENN G = x und x ≠ 0 und ziel (G = z)
DANN merke ("G = 0"), streiche ("G = x"), protokolliere ("Leere G.").
- (4) WENN E = x und x ≠ 0 und ziel (G = z)
DANN merke ("E = 0"), streiche ("E = x"), protokolliere ("Leere E.").
- (5) WENN G = x und E = y und x + y < Gmax und y > 0 und ziel (G = z)
DANN merke ("G = x + y", "E = 0"), streiche ("G = x", "E = y"),
protokolliere ("Umfüllen von E nach G.").
- (6) WENN G = x und E = y und x + y ≥ Gmax und y > 0 und x < Gmax und
ziel (G = z)
DANN merke ("G = Gmax", "E = y - Gmax + x"),
streiche ("G = x", "E = y"),
protokolliere ("Umfüllen von E nach G.").
- (7) WENN G = x und E = y und x + y < Emax und x > 0 und ziel (G = z)
DANN merke ("E = x + y", "G = 0"), streiche ("G = x", "E = y"),
protokolliere ("Umfüllen von G nach E.").
- (8) WENN G = x und E = y und x + y ≥ Emax und x > 0 und y < Emax und
ziel (G = z)
DANN merke ("E = Emax", "G = x - Emax + y"),
streiche ("G = x", "E = y"),
protokolliere ("Umfüllen von G nach E.").
- (9) WENN ziel (G = x) und G = x DANN streiche ("ziel (G = x)").

Wie interpretiert nun ein *Interpreter* ein solches WENN-DANN-Schema? Er vergleicht die hinter WENN stehenden Bedingungen mit den Notierungen auf dem Zettel, der die "Welt" beschreibt, und versucht, diese zur "Deckung" zu bringen. Dazu darf man die Variablen (hier: x, y, z), die in der Bedingung stehen, durch aktuelle Werte ersetzen, die man im Falle einer Übereinstimmung ("*matching*") mit den gleichen Werten in den Aktionen, die hinter DANN stehen, belegen muß. Vergleicht man zum Beispiel die "Welt", wie oben angegeben ist, mit der Regel (1), so kann

man eine Übereinstimmung erzielen, falls man z durch (3) ersetzt. Da Übereinstimmung vorliegt, darf man die Aktionen der Regel (1) ausführen:

merke bedeutet: Schreibe das, was in Klammern steht, auf den Zettel.
 streiche bedeutet: Entferne das, was in Klammern steht, vom Zettel.
 protokolliere bedeutet: Schreibe das, was in Klammern steht, ans Ende einer gesonderten Stelle des Zettels (dort werden die ausgeführten Handlungen protokolliert).

Man darf also zu Beginn den *Aktionsteil* der Regel (1)
 merke ("G = Gmax"), streiche ("G = 0"), protokolliere ("Fülle G.")
 ausführen. Der Zettelinhalt verändert sich also wie folgt:

Alte Weltbeschreibung		Neue Weltbeschreibung	Handlungsprotokoll
G = 0		E = 0	Fülle G.
E = 0	==>	ziel (G = 3)	
ziel (G = 3)		G = 7	

Man hätte auch die Regel (2) anwenden können; die "Welt" wäre dann folgendermaßen verändert worden:

G = 0		G = 0	Fülle E.
E = 0	==>	ziel (G = 3)	
ziel (G = 3)		E = 10	

Wendet man nun Regel (1) auf diese letzte Zettelschrift an, so erhält man:

G = 0		ziel (G = 3)	Fülle E.
ziel (G = 3)	==>	E = 10	Fülle G.
E = 10		G = 7	

Nun kann man nur noch Regel (3) oder (4) anwenden. Der WENN-Teil von Regel (3)

$G = x$ und $x \neq 0$ und $\text{ziel}(G = z)$

führt mit den Zuordnungen $x = 7$ und $z = 3$ zur Übereinstimmung; also kann der Aktionsteil merke ("G = 0"), streiche ("G = x"), protokolliere ("Leere G") mit den gleichen Variablenzuordnungen ausgeführt werden, und man erhält:

ziel (G = 3)	Fülle E.
E = 10	Fülle G.
G = 0	Leere G.

Nun kann man wieder Regel (1) oder die Regel (6) anwenden usw. Man sieht: Rechts erscheint die Folge der ausgeführten Handlungen untereinander, links steht die Beschreibung der augenblicklichen Situation. Das *Regelsystem* kann solange auf dem Zettel der Weltbeschreibung arbeiten, bis "ziel (G = 3)" mit Regel (9) entfernt wird. In unserem Fall erreicht man dies am schnellsten, wenn man die Regeln (2), (6), (3), (6) und (9) in dieser Reihenfolge anwendet, was die Folge

Fülle E. Umfüllen von E nach G. Leere G. Umfüllen von E nach G.
liefert. Man kann nun weitere Regeln hinzufügen, z.B.

(10) WENN Fülle G. · Leere G. DANN streiche ("Fülle G." · "Leere G.")

Hierbei soll '.' bedeuten, daß die beiden Textstücke direkt aufeinander folgen müssen. Diese Regel kann im Nachhinein sich gegenseitig aufhebende Handlungen wieder entfernen.

Das besondere von solchen Regelsystemen ist der *Nichtdeterminismus*, der uns bereits beim logik-basierten Programmieren begegnete: Man kann die Regeln in irgendeiner beliebigen Reihenfolge anwenden, wobei kein Hinweis existiert, *ob* und *wie* das anzusteuernde Ergebnis ("Ziel") zu erreichen ist. In der Praxis müssen solche Regelsysteme daher mit *Strategien* versehen werden, nach denen die einzelnen Regeln angewendet werden.

5.2 Übersicht über Programmiersprachen

Um aus der akademischen Sicht nicht den Blick für die industrielle Programmierpraxis zu verlieren, veranschaulicht Abb. 5.1 die relative Verbreitung von Programmiersprachen (das Maß ist der personelle Aufwand für Erstellung und Wartung von Programmen in der entsprechenden Programmiersprache).

Abb. 5.1 Verbreitung von Programmiersprachen

Wenn auch die zugrundeliegenden Zahlen - wie so vieles in der Informatik - schnell an Aktualität verlieren, zeigen sie dennoch eine ernüchternde Bilanz: Mit "niedrigen" imperativen Sprachen wird immer noch mindestens 3/4 der Programmierkapazitäten abgedeckt, forschungsrelevante Sprachen wie PROLOG oder LISP sind nahezu bedeutungslos.

Abb. 5.2 zeigt - entlang einer Zeitachse von 1950 bis 1988 - einige bekannte Programmiersprachen und idelle Abhängigkeiten zwischen ihnen (ein Pfeil " $x \rightarrow y$ " bedeutet, die Entwicklung der Sprache y wurde von der Sprache x beeinflusst). Da es inzwischen über tausend Programmiersprachen gibt, ist die *Auswahl* teilweise willkürlich, und die Abhängigkeiten zwischen den Sprachen sind unvollständig und im Einzelfall diskussionswürdig.

Abb. 5.2 Entwicklung von Programmiersprachen

Bei der Entwicklung der Programmiersprachen sind u.a. folgende Aspekte zu beobachten:

- **Abstrahierung:** Man geht zu immer mächtigeren, anwendungsnäheren Sprachen über, mit denen man mit wenig Aufwand immer kompliziertere Zusammenhänge bearbeiten kann. Die Sprachelemente orientiert man dabei an den zu bearbeitenden Problem- und Anwendungsbereichen. Beispiele: mathematisch-naturwissenschaftlich-technischer Bereich, Bürosysteme, Expertensysteme, Prototypenerstellung usw.
- **Struktur der Sprachen:** Waren früher Programme nur Befehlsfolgen, so müssen sie in Zukunft klare Strukturen aufweisen, d.h. z.B. eine klare Trennung zwischen "Wissen" und "Handeln". Modularisierung, Objektorientierung, Gliederung bzgl. Parallelität, Kommunikationsstrukturen und Zugriff auf anderweitiges Wissen

(Daten- und Methodenbanken) sind weitere Struktureigenschaften bei der Entwicklung von Sprachen.

- *Sicherheit*: Dies betrifft zum einen die Fehlerfreiheit und Robustheit der Programme, die in Software-Entwicklungsumgebungen getestet oder - in einigen Fällen - bewiesen werden kann, zum anderen auch die aufgabengerechte Verwendung der Programme in großen Programmsystemen.
- *Rechnerarchitektur*: Aus Effizienzgründen richtet man viele Sprachen nach den jeweils vorhandenen Rechnern aus. In den letzten Jahren hat es aber auch das Umgekehrte gegeben (z.B. spezielle LISP- oder PROLOG-Maschinen).
- *Benutzungsoberfläche, Mensch-Maschine-Schnittstelle*: Stichworte sind hier Dialogsysteme, Kommunikationsunterstützung, Graphik, "gewohnte" Umgebung, leichte Erlernbarkeit usw.

Denken vollzieht sich im Rahmen einer Sprache. Die jeweilige Programmiersprache legt die Denkwelt fest, in der man Probleme mit Hilfe von Rechnern löst. Wer mit Rechnern umgeht, sollte sich nicht auf *eine* Sprachwelt festlegen; sie wird ihn nur behindern, die nächste Generation von Rechnern, ihren Sprachen und Systemen benutzen und verstehen zu können.

6. Datenstrukturen und Algorithmen

In den vorausgegangenen Kapiteln haben wir uns “nur” um die Konstruktion effektiver und sauber strukturierter Programme bemüht. Im Vordergrund standen also qualitative, funktionale Aspekte des Algorithmenentwurfs, ohne *quantitative* Aspekte, also z.B. ohne die *Effizienz* von Programmen zu betrachten.

Aber gerade um die Effizienz geht es in diesem Kapitel, das sich vorrangig mit Algorithmen zum Suchen und Sortieren sowie zur Speicherverwaltung beschäftigt. Irgendein Such- oder Sortiervorgang ist oft schnell gefunden, das jedoch bei großen zu durchsuchenden oder zu sortierenden Datenbeständen inakzeptabel lange dauert oder (allerdings seltener) unverträglich viel Speicherplatz beansprucht.

Deshalb soll als Vorbereitung auf die spätere *Analyse* von Algorithmen in Abschnitt 6.1 das Thema “Effizienz von Algorithmen” behandelt werden, das in unmittelbarem Zusammenhang mit der *Komplexität* von Funktionen steht. “Komplex” hat hier nicht die umgangssprachliche Bedeutung “kompliziert”, sondern ist die formale Definition von minimalem oder durchschnittlichem “Aufwand zur Berechnung”.

In Abschnitt 6.2 stellen wir *Graphen* und *Bäume* als universelle rekursive Datenstrukturen vor, die das *Sortieren* von Elementfolgen (Abschnitt 6.4) und das *Suchen* in gegebenen (Abschnitt 6.3) oder dynamisch sich ändernden (Abschnitt 6.5) Datenbeständen effizient unterstützen können. Abschnitt 6.6 widmet sich schließlich algorithmischen Konzepten zur *Speicherverwaltung*.

6.1 Effizienz und Komplexität

6.1.1 Motivation und Begriffsbildung

Effizienz und Komplexität sind nicht trennbare, auf die *Bewertung* von Algorithmen zielende Begriffe. Bewertungskriterien sind die durch Algorithmen in Anspruch genommenen *Ressourcen* wie *Zeit* und *Speicher*.

Dabei bezieht sich die Komplexität auf eine *Funktion*, die Effizienz hingegen auf einen konkreten *Algorithmus* (der eine Funktion realisiert). Typischerweise schätzt die Komplexität durch eine *untere* Schranke die Effizienz von Algorithmen “von unten her” ab, d.h. alle eine Funktion realisierenden Algorithmen beanspruchen *mindestens* den durch die untere Schranke vorgegebenen Aufwand. Umgekehrt ist die Effizienz

eines Algorithmus immer *obere* Schranke für die Komplexität der durch den Algorithmus realisierten Funktion.

Von einem eher praktischen Standpunkt versucht man immer effizientere Algorithmen zu finden, von einem eher theoretischen Standpunkt zielt man auf immer exaktere (d.h. höhere) untere Schranken. Beide Vorhaben treffen sich in dem Bestreben, die Lücke zwischen den effizientesten Algorithmen und der Komplexität der dadurch realisierten Funktion möglichst klein werden zu lassen.

Das Beschleunigen von Algorithmen wird uns in den späteren Abschnitten mehr interessieren. Wir lernen typische Verfahren zur Effizienzsteigerung (z.B. "Teile und Herrsche") kennen. Doch sollen nachfolgend nicht nur die Grundlagen zur Berechnung der Effizienz von Algorithmen gelegt (Abschnitt 6.1.2), sondern auch einige Elemente der *Komplexitätstheorie* behandelt werden (Abschnitt 6.1.3).

6.1.2 Effizienz von Algorithmen

Wir interessieren uns zunächst für die Effizienz von Algorithmen in Form von Programmen einer Programmiersprache. Dabei sollte man nicht vergessen, daß die Effizienz mit anderen Eigenschaften, beispielsweise mit der Lesbarkeit und mit der Portabilität *konkurriert*. Global gesehen geht es bei der Programmerstellung immer um eine *Minimierung* der Kosten (oder um eine Maximierung des Nutzens). Die Effizienz spielt dabei i. a. keine herausragende Rolle, "schneller" und "besser" sind keinesfalls synonym.

Allerdings gibt es Algorithmen, die sehr oft ausgeführt werden und daher starken Einfluß auf die Leistung eines Rechners haben, beispielsweise im Kern des Betriebssystems, im Editor oder im Übersetzer. Für solche zentralen Programme lohnt sich die sorgfältige Wahl effizienter Algorithmen allemal.

Beim Vergleich verschiedener Algorithmen will man eine möglichst *allgemeine* Antwort erhalten, also eine, die vom eingesetzten Rechner und von der Repräsentation der Daten möglichst unabhängig ist.

Verschiedene Methoden der Effizienzbeurteilung

Solange Rechner und Daten festliegen (siehe Zeile 1 in Abb. 6.1), genügt eine *Zeitmessung* (mit der "eingebauten Uhr" des Rechners oder auch von Hand). *Zählt* man stattdessen die einzelnen Operationen, so kann man vom Rechner, oder allgemeiner vom zugrundeliegenden Maschinenmodell, abstrahieren (Zeile 2 in Abb. 6.1): der konkrete Zeitbedarf ergibt sich dann wieder durch Berücksichtigung der Ausführungszeiten auf einem konkreten Maschinenmodell oder Rechner.

Um auch von den Eingabedaten abstrahieren zu können (Zeile 3 in Abb. 6.1), *berechnet* man die Zahl notwendiger Operationen, statt sie zu zählen. Dabei gehen in

der Regel Parameter ein, die die Größe des Problems beschreiben (z.B. die Anzahl der Elemente bei einem Sortierproblem). Damit hat man dann eine *abstrakte* Beschreibungsform gefunden, um Algorithmen zu vergleichen. Das Resultat des Vergleichs hängt aber noch immer von der konkreten Parametrisierung ab.

Rechner/Maschine	Eingabedaten	Methode der Beurteilung
festgelegt	festgelegt	Zeitmessung
abstrakt	festgelegt	Zählung
abstrakt	abstrakt	Berechnung
abstrakt	$\rightarrow \infty$	asymptotische Abschätzung

Abb. 6.1 Methoden der Effizienzbeurteilung

Frage: Sei V die Zahl der Vergleiche und W die der Wertzuweisungen in einem Algorithmus, der eine Datenstruktur der Größe n bearbeitet. Für zwei Algorithmen A_1 und A_2 gelte:

Aufwand für A_1 : $V_1 = W_1 = n \cdot \log(n)$, Aufwand für A_2 : $V_2 = n^2$, $W_2 = n$.

Welcher der Algorithmen ist schneller?

Offenbar kann die Frage nur mit Kenntnis von n und der Ausführungszeiten für Vergleich und Wertzuweisung beantwortet werden. Aber auch ohne diese Daten läßt sich *eine* Aussage sicher treffen: Übersteigt n einen "break-even point", so ist der Algorithmus A_1 schneller, denn sein Zeitbedarf wächst nur proportional $n \cdot \log(n)$, nicht proportional n^2 . Damit ist die Abstraktionsstufe der *asymptotischen Aufwandsabschätzung* erreicht (Zeile 4 in Abb. 6.1), die später noch ausführlich behandelt wird.

Experimentell läßt sich der Vergleich von Algorithmen nicht durchführen. Sei beispielsweise der Rechenaufwand eines Algorithmus in Abhängigkeit einer Felddimension n beschrieben durch $T(n) = 100n + n^2 + 10n/1000-2$.

In diesem Fall dominiert für Werte unter 100 der lineare Summand, darüber zunächst der quadratische; erst bei sehr hohen Werten (über 10.000) wird der dramatische Effekt der exponentiellen Komponente deutlich. Woher soll man ohne Kenntnis der Funktion $T(n)$ wissen, bis zu welchen Werten von n zu messen ist? Eine solche Frage kann also nur nach einer *Analyse* des Algorithmus beantwortet werden.

Programme enthalten i.a. verschiedene Operationen, deren Ausführungszeiten variieren, vielfach um Größenordnungen. Beispielsweise kann eine REAL-Division so aufwendig wie die Summe vieler INTEGER-Operationen sein. Bei der Aufwandsanalyse

sind vor allem zu berücksichtigen (und jeweils mit den Typen der Daten zu gewichten):

- Wertzuweisungen
- Vergleiche
- Rechenoperationen
- Prozeduraufrufe und Sprünge im Maschinencode.

Die Organisation von Schleifen erfordert pro Durchlauf typisch je einen Vergleich, eine Addition, eine Wertzuweisung und einen Sprung.

In den weiteren Abschnitten dieses Kapitels enthalten die Algorithmen vor allem *Such-* und *Verschiebeoperationen* im Speicher, so daß dort zwei Operationstypen dominieren, nämlich Vergleiche und Wertzuweisungen. Bei beiden hängt der Aufwand natürlich noch stark von den Typen ab: der Vergleich zweier natürlicher Zahlen oder eine Wertzuweisung zwischen einzelnen Zeichen ist sehr "billig", der Vergleich von Zeichenreihen oder die Wertzuweisung von Records kann sehr "teuer" sein.

Die Frage, wie sich die Rechenzeit mit den Daten *prinzipiell* ändert, wird, wie oben bereits ausgeführt, durch *asymptotische Abschätzung* beantwortet, d.h. das am schnellsten wachsende Glied der Funktion $T(n)$ wird bestimmt. Im Beispiel oben wäre dies das letzte Glied; da man nur an der Art der Funktion, nicht an konstanten Faktoren interessiert ist, kann man sagen, daß T von der Ordnung $10^{n/1000}$ ist, denn die "-2" im Exponenten entspricht einem konstanten Faktor $1/100$. Im Vergleich zu den anderen Summanden genügt sogar die Aussage, daß T exponentiell wächst, denn dieses Wachstum "schlägt" alle Terme mit polynomialem Wachstum.

Im folgenden werden wir die bisherigen Effizienzbetrachtungen bezüglich asymptotischer Abschätzungen formaler fassen und damit präzisieren.

Asymptotische Abschätzung

Die von einem Algorithmus verarbeitbaren Eingaben bilden eine *Sprache*. Aus diesem Grund werden die weiteren Aussagen über die Effizienz von Algorithmen auf die Verarbeitung von Sprachen bezogen. Eingaben sind dann Wörter über einem der Sprache zugrundeliegenden Alphabet Σ .

Sei A ein Algorithmus, dessen elementare Anweisungen und Bedingungen höchstens eine *Zeiteinheit* zur Ausführung benötigen (Abstrahierung von Maschine und konkreten Sprachelementen). Wir betrachten hier nur Algorithmen A , die für jede Eingabe terminieren.

Die von A verwendete *Eingabe* sei ein Element aus Σ^* für ein Alphabet Σ . Für jeden Eingabewert $w \in \Sigma^*$ sei

$s(w)$ = Anzahl der Schritte, die der Algorithmus A bei Eingabe des Wortes w benötigt, bis er anhält.

Setze $s_A : N_0 \rightarrow N_0$ mit
 $s_A(n) = \max \{s(w) \mid w \in \Sigma^*, |w| = n\}.$

$s_A(n)$ gibt also die *maximale Schrittzahl* an, die zur Verarbeitung eines Wortes der Länge n vom Algorithmus A benötigt wird.

Die maximale Schrittzahl ist das übliche Effizienzmaß bei der Analyse von Algorithmen: man

- interessiert sich für die *Laufzeit*,
- möchte nur eine Aussage über den *schlechtesten*, d.h. am längsten dauernden Fall,
- berücksichtigt *nicht* Details der zugrundeliegenden Programmiersprache bzw. der Maschine, auf der der Algorithmus abläuft.

Mit diesem Effizienzbegriff werden also drei Annahmen getroffen:

- (a) Effizienz wird durch *Zeitkomplexität* (Schritzahl) definiert,
- (b) man betrachtet eine *obere Schranke* (max-Bildung),
- (c) Komplexität ist weitgehend *unabhängig von der Zielmaschine*.

zu (a) Man kann auch *Speicherkomplexität* definieren:

$p(w)$ = Anzahl der Speicherplätze, die A bei Eingabe des Wortes w benötigt.

$p_A : N_0 \rightarrow N_0$ mit

$p_A(n) = \max \{p(w) \mid w \in \Sigma^*, |w| = n\}.$

zu (b) Man kann neben einer oberen auch eine untere Schranke definieren und eine relativ "exakte" Effizienzabschätzung versuchen. Dies ist jedoch oft sehr schwierig, da komplizierte wahrscheinlichkeitstheoretische Untersuchungen notwendig sind.

Sei $f : N_0 \rightarrow N_0$. Dann sind folgende Mengen von Funktionen

$O(f) = \{g : N_0 \rightarrow N_0 \mid \exists c > 0 \exists n_0 \text{ mit}$
 $f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$ *obere Schranken,*

$U(f) = \{g : N_0 \rightarrow N_0 \mid \exists c > 0 \exists n_0 \text{ mit}$
 $f(n) \geq c \cdot g(n) \text{ für alle } n \geq n_0\}$ *untere Schranken,*

$EX(f) = \{g : N_0 \rightarrow N_0 \mid \exists c > 0 \exists n_0 \text{ mit}$
 $1/c \cdot g(n) \leq f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0\}$ *exakte Schranken.*

Praktisch heißt das z.B., daß eine Funktion g genau dann eine obere Schranke von f ist, wenn von irgendeinem definierten Wert für n an f nicht über $c \cdot g$ steigt (für eine positive Konstante c).

- zu (c) Man kann *maschinen- oder sprachabhängige* Komplexitätsuntersuchungen durchführen, indem man z.B. Turing-Maschinen (deterministische und nicht-deterministische) oder die konkrete Programmiersprache mit Aussagen über deren Elementaranweisungen auf konkreten Datenstrukturen mit relativem oder sogar absolutem Zeitbedarf zugrundelegt.

Definition von Zeitkomplexitätsklassen

Ein Algorithmus A lese ein Wort $w \in \Sigma^*$ ein und gebe ein Wort $v \in \Delta^*$ aus. A berechnet also eine Funktion $f_A : \Sigma^* \rightarrow \Delta^*$. Wir setzen voraus, daß A für alle Eingaben anhält. Man unterscheidet i.a. folgende drei zentralen *Zeitkomplexitätsklassen*:

A heißt *linear-zeitbeschränkt*, wenn $s_A \in O(n)$

(d.h. $\exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : s_A(n) \leq c \cdot n$).

A heißt *polynomial-zeitbeschränkt*, wenn ein $k \in \mathbb{N}$ existiert, so daß $s_A \in O(n^k)$

(d.h. $\exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : s_A(n) \leq c \cdot n^k$).

A heißt *exponentiell-zeitbeschränkt*, wenn ein $k \in \mathbb{N}$ existiert, so daß $s_A \in O(k^n)$

(d.h. $\exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : s_A(n) \leq c \cdot k^n$).

Von besonderem Interesse, da in dieser Klasse viele in der Praxis interessierende Algorithmen liegen, sind die polynomial-zeitbeschränkten Algorithmen.

Nichtdeterminismus

Man nennt einen Algorithmus A *nichtdeterministisch*, wenn A das folgende Sprach-element *OR* (in beliebiger Anzahl) enthält:

α *OR* β soll bedeuten, daß man entweder mit der Anweisung α oder mit der Anweisung β fortfahren kann, wenn man bei der Abarbeitung eines Algorithmus auf ein solches Sprachelement stößt. Die Auswahl zwischen den beiden Alternativen ist dabei *willkürlich*.

Anschaulich kann man sich die Ausführung nichtdeterministischer Algorithmen so vorstellen, daß bei mehreren Fortsetzungsmöglichkeiten stets die richtige "*geraten*" wird. Mit Hilfe eines solchen - natürlich fiktiven - Vorgehens könnte man beispielsweise den Weg zum Ausgang eines Labyrinths in der kürzest möglichen Zeit finden.

Weiterhin möge der Algorithmus A ohne Beschränkung der Allgemeinheit nur das Ergebnis TRUE oder FALSE liefern, also wie eine *boolesche Funktion* arbeiten. Es interessieren nun "kürzeste" Berechnungen solcher Algorithmen, wozu wieder die Schrittzahlfunktion herangezogen werden kann.

Für $w \in \Sigma^*$ sei

$$s(w) = \begin{cases} \text{die minimale Anzahl von Schritten, in der A bei Eingabe von } w \text{ und} \\ \text{“geschickter” Auswahl an jedem OR TRUE liefert, bzw.} \\ 0, \text{ falls TRUE nicht erzeugt werden kann.} \end{cases}$$

Diese Definition der Schrittzahlfunktion s ist nur für die Wörter $w \in \Sigma^*$ sinnvoll, für die der Algorithmus A für irgendeine Auswahl an den OR-Alternativen den Wert TRUE liefert. Die Wörter, die stets FALSE liefern, werden in der Schrittzahlfunktion *im Gegensatz zum deterministischen Fall* nicht berücksichtigt. (Dies hat Konsequenzen für das Komplement von Sprachen in NP, siehe unten.)

Man verwechsle das Denkmodell des nichtdeterministischen Algorithmus nicht mit einer Realisierungsidee; als Denkmodell benötigen wir es, weil es offenbar von realen Algorithmen nicht übertroffen werden kann und darum eine sichere obere Schranke bezüglich des Rechenaufwandes darstellt.

Auch für nichtdeterministische Algorithmen abstrahieren wir vom konkreten Eingabewort und gehen zur *maximalen Schrittzahl* für gleichlange Worte über:

$$s_A(n) = \max \{s(w) \mid w \in \Sigma^* \text{ und } |w| = n\}.$$

Wir betrachten im folgenden wieder polynomial-zeitbeschränkte Algorithmen, die auch im nichtdeterministischen Fall interessanter als die linear- oder exponentiell-zeitbeschränkten sind.

Ein nichtdeterministischer Algorithmus A heißt *nichtdeterministisch polynomial-zeitbeschränkt*, wenn es ein $k \in \mathbb{N}$ mit $s_A \in O(n^k)$ gibt.

6.1.3 Komplexität von Funktionen und Sprachen

Oft abstrahiert man von einem konkreten Algorithmus (der ja sehr “ungeschickt”, d.h. in dieser Beziehung sehr langsam realisiert sein kann) und geht zur Charakterisierung der Komplexität von *Funktionen* über.

Eine (totale) Funktion $f : \Sigma^* \rightarrow \Delta^*$ heißt *polynomial-zeitberechenbar* (pzb) : $\Leftrightarrow$ es gibt einen polynomial-zeitbeschränkten Algorithmus A mit $f = f_A$.

In einem weiteren Schritt kann man nun die Komplexität von Sprachen charakterisieren, indem man als Kriterium den Aufwand für die Entscheidung nimmt, ob ein Wort zu dieser Sprache gehört oder nicht.

Es sei Σ ein Alphabet und $L \subseteq \Sigma^*$.

$C_L : \Sigma^* \rightarrow \{0, 1\}$ mit $C_L(w) = 1$, falls $w \in L$, bzw. 0, falls $w \notin L$

heißt *charakteristische Funktion* von L .

L heißt *pzb-Sprache* wenn C_L pzb ist.

pzb-Sprachen und -Funktionen

$P := \{ L \subseteq \Sigma^* \mid L \text{ ist pzb und } \Sigma \text{ ist ein Alphabet} \}$ heißt *Klasse der pzb-Sprachen*,

$P := \{ f : \Sigma^* \rightarrow \Delta^* \mid f \text{ ist pzb und } \Sigma, \Delta \text{ sind Alphabete} \}$ heißt *Klasse der pzb-Funktionen*.

Nichtdeterminismus

Eine Sprache L heißt *nichtdeterministisch polynomial-zeitberechenbar* (npzb), wenn C_L von einem nichtdeterministisch polynomial-zeitbeschränkten Algorithmus berechnet wird.

Genauer: ..., wenn es einen nichtdeterministischen Algorithmus A gibt mit

- $\forall w \in L$ gilt: Es gibt eine geschickte Auswahl an jedem OR, so daß A bei Eingabe von w TRUE ausgibt.
- $\forall w \notin L$ gilt: Für jede Auswahl an jedem OR gelangt der Algorithmus A bei Eingabe von w stets zur Ausgabeanweisung für FALSE.
- A ist nichtdeterministisch polynomial-zeitbeschränkt.

$NP := \{ L \subseteq \Sigma^* \mid L \text{ ist npzb und } \Sigma \text{ ist ein Alphabet} \}$ heißt *Klasse der npzb-Sprachen*.

Beispiel einer Sprache aus NP:

Sei $L = \{ w \in \{0, 1\}^* \mid \text{die durch (die Binärzahl) } w \text{ dargestellte Dezimalzahl } w \text{ ist keine Primzahl} \}$.

Wir wollen zeigen, daß $L \in NP$ gilt. Hierzu "raten" wir eine Dezimalzahl $v \neq 1$ mit $v < w$ und testen, ob v die Dezimalzahl w teilt. Falls ja, geben wir TRUE aus, sonst FALSE.

"Beweis" durch Angabe eines nichtdeterministisch polynomial-zeitbeschränkten Algorithmus in Pseudo-MODULA-2:

In einem beliebig langen Feld stehe nur die Binärzahl w .

(* Programm P6.1 *)

```

NONDETERMINISTIC PROCEDURE Nichtprim
    (Zahlw : ARRAY OF BOOLEAN) : BOOLEAN;
VAR i, w, v, rest : INTEGER;
BEGIN
    w := 0; v := 0;

```



```

FOR i := 0 TO HIGH (Zahlw) DO                (* Zahl w berechnen *)
  w := w * 2;
  IF Zahlw [i] THEN INC (w); END;
  (v := 2 * v) OR (v := 2 * v + 1);          (* Teiler v von w *)
END (* FOR *);                                (* raten *)
IF (v # 0) AND (v # 1) AND (v < w) THEN
  rest := w MOD v;                          (* Pruefung, ob v Teiler von w *)
ELSE
  rest := 1;
END (* IF *);
RETURN rest = 0;                             (* Ergebnis zurueckgeben *)
END Nichtprim;

```

Durch die FOR-Schleife wird irgendeine Dezimalzahl v zwischen 0 und 2^m-1 geraten. Die Ausgabe TRUE wird erreicht, wenn eine solche Zahl ($\neq 0$, $\neq 1$, $< w$) die eingeleseene Zahl teilt. Ist w nicht prim, so existiert mindestens eine solche Zahl, womit TRUE in $O(m) = O(|w|)$ Schritten erreicht wird. (Hierbei wird aber angenommen, daß die Restbildung MOD ebenfalls in $O(|w|)$ Schritten erfolgen kann.) Andernfalls kann TRUE nicht erreicht werden, womit die drei obigen Bedingungen für eine npzb-Sprache nachgewiesen sind. Es folgt also: (Nichtprim) $L \in NP$.

Bemerkungen:

1. Man beachte, daß hierdurch nichts über das Komplement $\text{Comp}(L)$ von L ausgesagt ist, also über die Sprache $\text{Comp}(L) = \{w \in \{0, 1\}^* \mid \text{die durch } w \text{ dargestellte Zahl } w \text{ ist eine Primzahl}\}$. Denn die Schrittzahlfunktion des obigen nichtdeterministischen Algorithmus in Programm P6.1 berücksichtigt nicht die Wörter, die in $\text{Comp}(L)$ liegen.
2. Bis heute ist ungelöst, ob L in P liegt oder nicht. Ein sehr bekanntes ungelöstes Problem der Theoretischen Informatik ist die Frage $P=NP?$ ("P-NP-Problem"). Die vermutete Antwort "Nein" konnte bis heute nicht bewiesen werden.

Problemreduktion

Häufig versucht man, eine (neue) Sprache (sprich ein Problem) auf eine (schon bekannte) andere Sprache (sprich ein anderes Problem) zu "*reduzieren*", dessen Komplexität man schon kennt, um damit auch die neue Sprache in ihrer Schwierigkeit einzuschätzen. In der interessanten Klasse der *NP-Sprachen* interessiert man sich besonders für die polynomial-zeitbeschränkte (man bleibt also in polynomialen Bereich!) *Reduzierbarkeit* auf bekannte Sprachen, die evtl. selbst in NP liegen.

Definitionen

1. $L, L' \subseteq \Sigma^*$ seien Sprachen.
 L' heißt mit *polynomialem Aufwand auf L reduzierbar* : $\Leftrightarrow$
es existiert eine pzb Funktion $f: \Sigma^* \rightarrow \Sigma^*$, so daß für alle $w \in \Sigma^*$ gilt:
 $(w \in L' \Leftrightarrow f(w) \in L)$.
Schreibweise hierfür: $L' \leq_p L$
2. Eine Sprache L heißt *NP-hart* (bzgl. $\leq_p$) : $\Leftrightarrow$
für jede Sprache $L' \in \text{NP}$ gilt: $L' \leq_p L$.
3. Eine Sprache L heißt *NP-vollständig* (bzgl. $\leq_p$) : $\Leftrightarrow$
 L ist NP-hart und $L \in \text{NP}$.

Man kennt heute über 1.000 NP-vollständige Sprachen ("Probleme"). Viele von ihnen (z.B. Transport- und Logistikprobleme) sind für die Praxis von großer Bedeutung. Ein sehr anschauliches NP-vollständiges Problem ist z.B. das *Rucksackproblem*:

Gegeben ist eine Folge von natürlichen Zahlen $a_1, a_2, a_3, \dots, a_n, b$.

Gesucht ist eine Indexmenge $I \subseteq \{1, 2, \dots, n\}$ mit $\sum_{i \in I} a_i = c \leq b$ und $b-c$ ist minimal (anschaulich "Packe den Rucksack mit Volumen b mit den Gegenständen a_i möglichst voll").

Beispiel:

Gegeben sei das Rucksackproblem mit folgenden $n = 8$ Zahlen: $a_1 = 17, a_2 = 26, a_3 = 28, a_4 = 36, a_5 = 45, a_6 = 46, a_7 = 52, a_8 = 57$ und das Volumen $b = 195$. Durch Ausprobieren stellt man als maximales $c \leq b$ fest: $c = 193 = 17 + 28 + 45 + 46 + 57$. Eine gesuchte Indexmenge lautet also $I = \{1, 3, 5, 6, 8\}$.

6.2 Graphen und Bäume

Graphen und (als Spezialfall) Bäume sind sehr universelle Strukturen/Modelle, mit denen sich viele Probleme und Methoden in der Informatik beschreiben lassen. Anschaulich bestehen *Graphen* aus Punkten (*Knoten* genannt), die durch Linien (*Kanten* genannt) verbunden sind. Je nachdem, ob diese Kanten eine Richtung (*Orientierung*) haben, spricht man von gerichteten oder von ungerichteten Graphen.

Bäume (zusammenhängende Graphen, die keine zyklischen Linienverbindungen enthalten dürfen) werden später in den Abschnitten über *Suchen* und *Sortieren* als für solche Algorithmen häufig geeignete Datenstrukturen sehr intensiv verwendet.

Der Abschnitt 6.2.1 beschränkt sich auf die notwendigen Definitionen und enthält bis auf das Beispiel des Auffindens sogenannter starker Zusammenhangskomponenten in Graphen nur wenige algorithmische Elemente.

6.2.1 Graphen

1. $G = (V, E)$ heißt *ungerichteter Graph* : $\Leftrightarrow$
 - (i) $V \neq \emptyset$ ist endliche Menge. V heißt *Knotenmenge*, Elemente von V heißen *Knoten*.
 - (ii) E ist eine Menge ein- und zweielementiger Teilmengen von V . E heißt *Kantenmenge*, ein Paar $\{u, v\} \in E$ heißt *Kante*. Eine Kante $\{u\}$ heißt *Schlinge*.

Zwei Knoten u und v heißen *benachbart* : $\Leftrightarrow \{u, v\} \in E \vee (u = v \wedge \{u\} \in E)$.
Durch die verbindende Kante werden Knoten also zu Nachbarn, durch eine Schlinge auch von sich selbst.

2. Sei $G = (V, E)$ ein ungerichteter Graph. Wenn E keine Schlingen enthält, so heißt G *schlingenlos*.
Seien $G = (V_G, E_G)$ und $H = (V_H, E_H)$ ungerichtete Graphen.
 H heißt *Teilgraph* von G ($H \subset G$): $\Leftrightarrow V_H \subset V_G$ und $E_H \subset E_G$.
 H heißt *vollständiger Teilgraph* von G : $\Leftrightarrow$

$$H \subset G \text{ und } [(u, v) \in E_G \text{ mit } u, v \in V_H \Rightarrow (u, v) \in E_H].$$

In H fehlen also nur diejenigen Kanten aus G , zu denen Endknoten fehlen.

Im weiteren werden wir Kanten $\{u, v\}$ als Paare (u, v) oder (v, u) und Schlingen $\{u\}$ als Paar (u, u) schreiben, um später nicht Definitionen für ungerichtete und gerichtete Graphen differenzieren und notationell unterscheiden zu müssen.

Beispiel 1 (vgl. Abb. 6.2)

$G = (V_G, E_G)$ mit $V_G = \{1, 2, 3, 4\}$ und
 $E_G = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$.

Abb. 6.2 Graphische Repräsentierungen von G aus Beispiel 1

3. $G = (V, E)$ heißt *gerichteter Graph* : $\Leftrightarrow$
- (i) $V \neq \emptyset$ ist endliche Menge. V heißt Knotenmenge, Elemente von V heißen Knoten.
 - (ii) $E \subseteq V \times V$ heißt Kantenmenge. Elemente von E heißen Kanten.
 Schreibweise: (u, v) oder $u \rightarrow v$.
 u ist die *Quelle*, v das *Ziel* der Kante $u \rightarrow v$.
 Eine Kante (u, u) heißt Schlinge.

Beispiel 2 (vgl. Abb. 6.3)

$G = (V_G, E_G)$ mit $V_G = \{1, 2, 3, 4\}$ und
 $E_G = \{1 \rightarrow 2, 1 \rightarrow 3, 1 \rightarrow 4, 2 \rightarrow 3, 2 \rightarrow 4, 3 \rightarrow 4\}$

Abb. 6.3 Darstellung von G aus Beispiel 2

4. Sei $G = (V_G, E_G)$ ein (un)gerichteter Graph und $G' = (V_{G'}, E_{G'})$ mit $V_{G'} \subset V_G$ und $E_{G'} \subset E_G$ (G' muß also kein Graph sein!).
 $H = (V_H, E_H)$ entsteht aus G durch Weglassen von G' , wenn
 $V_H = V_G \setminus V_{G'}$ und
 $E_H = \{(u, v) \mid (u, v) \in E_G \setminus E_{G'} \text{ und } u, v \in V_H\}$.
 Schreibweise: $H = G - G'$.

5. Seien $G = (V_G, E_G)$ und $G' = (V_{G'}, E_{G'})$ (un)gerichtete Graphen.
 Der Graph $H = (V_H, E_H)$ entsteht aus G durch Hinzufügen von G' , wenn
 $V_H = V_G \cup V_{G'}$ und $E_H = E_G \cup E_{G'}$ ist.
 Schreibweise: $H = G + G'$.

Sei für die folgenden Punkte 6. bis 9. $G = (V, E)$ ein (un)gerichteter Graph und
 $k = (v_0, \dots, v_n) \in V^{n+1}$ eine Folge von $n+1$ Knoten.

6. k heißt *Kantenfolge* der Länge n (also $|k| = n$) von v_0 nach v_n , wenn für alle $i \in \{0, \dots, n-1\}$ gilt: $(v_i, v_{i+1}) \in E$.
 Im gerichteten Graphen ist v_0 der *Startknoten* und v_n der *Endknoten*, im ungerichteten Graphen sind v_0 und v_n die Endknoten von k .
 $v_1, \dots, v_{n-1}$ sind die *inneren Knoten* von k .
 Ist $v_0 = v_n$, so ist die Kantenfolge *geschlossen*.
7. k heißt *Kantenzug* der Länge n von v_0 nach v_n , wenn k Kantenfolge der Länge n von v_0 nach v_n ist und wenn für alle $i, j \in \{0, \dots, n-1\}$ mit $i \neq j$ gilt: $(v_i, v_{i+1}) \neq (v_j, v_{j+1})$. Im Kantenzug kommt also keine Kante mehrfach vor.
8. k heißt *Weg* der Länge n von v_0 nach v_n , wenn k Kantenfolge der Länge n von v_0 nach v_n ist und wenn für alle $i, j \in \{0, \dots, n\}$ mit $i \neq j$ gilt: $v_i \neq v_j$. Im Weg kommt also kein Knoten mehrfach vor.
9. k heißt *Zyklus* oder *Kreis* der Länge n , wenn k geschlossene Kantenfolge der Länge n von v_0 nach v_n und wenn $k' = (v_0, \dots, v_{n-1})$ ein Weg ist.
 Ein Graph ohne Zyklus heißt *kreisfrei*, *zyklenfrei* oder *azyklisch*.

Beispiel 3 (vgl. Abb. 6.4)

- $(1, 2, 3, 4, 5, 2, 3)$ ist Kantenfolge (aber nicht Kantenzug) der Länge 6 von 1 nach 3.
 $(1, 2, 5, 4, 2, 3)$ ist Kantenzug (aber nicht Weg) der Länge 5 von 1 nach 3.
 $(1, 2, 5, 4, 3)$ ist Weg der Länge 4 von 1 nach 3.
 $(2, 3, 4, 5, 2)$ ist Zyklus der Länge 4.

Abb. 6.4 Graph G für Beispiel 3

10. Sei $G = (V, E)$ ein ungerichteter Graph.

G heißt *zusammenhängend* : $\Leftrightarrow$

$\forall v, w \in V$: es existiert eine Kantenfolge k von v nach w , d.h. $k = (v, \dots, w)$ ist Kantenfolge in G , es sind also alle Knoten miteinander verbunden.

Ein zusammenhängender vollständiger Teilgraph T heißt *Zusammenhangskomponente* oder *Komponente* von G , wenn er maximal ist, d.h. es existiert kein Knoten $v \in G \wedge v \notin T$, so daß der um v erweiterte Graph T zusammenhängend und vollständig wäre.

Mit $C(G)$ bezeichnen wir die *Menge der Komponenten* von G .

11. Sei $G = (V, E)$ ein gerichteter Graph.

G heißt *zusammenhängend*, wenn der zu G gehörende ungerichtete Graph zusammenhängend ist, d.h. für alle $u, v \in V$ mit $u \neq v$ eine Kantenfolge $(v_0, v_1, \dots, v_n)$ existiert mit $u = v_0, v = v_n$ und für jedes i ($0 \leq i \leq n-1$) gilt: $(v_i, v_{i+1}) \in E$ oder $(v_{i+1}, v_i) \in E$.

Analog wird die Zusammenhangskomponente definiert.

G heißt *stark zusammenhängend*, wenn für alle $u, v \in V$ mit $u \neq v$ eine Kantenfolge von u nach v und von v nach u existiert.

Analog wird starke (Zusammenhangs)komponente definiert.

Beispiel 4 (vgl. Abb. 6.5)

G' ist zusammenhängend, aber nicht stark zusammenhängend.

G'' ist stark zusammenhängend.

Abb. 6.5 Graphen für Beispiel 4

12. Seien $G = (V, E)$ ein (un)gerichteter Graph, M_V und M_E Mengen und $f : V \rightarrow M_V$ und $g : E \rightarrow M_E$ Abbildungen.

$G' = (V, E, f)$ heißt *knotenmarkierter Graph*.

$G'' = (V, E, g)$ heißt *kantenmarkierter Graph*.

$G''' = (V, E, f, g)$ heißt *knoten- und kantenmarkierter Graph*.

M_V und M_E sind die *Markierungsmengen* (meistens Alphabete, freie Monoide oder Zahlen).

Wichtige algorithmische Probleme für Graphen

Wenn im folgenden keine Einschränkung vereinbart ist, sei $G = (V, E)$ ein (un)gerichteter Graph.

- Man entscheide, ob G *zusammenhängend* ist.
- Man entscheide, ob G *azyklisch* ist.
- Man entscheide, ob G einen *Hamiltonschen Zyklus* besitzt, d.h. einen Zyklus der Länge $|V|$, also über alle Knoten.
- Man entscheide, ob G einen *Eulerschen Weg* besitzt, d.h. eine Kantenfolge, in dem jede Kante genau einmal verwendet wird, und dessen Anfangs- und Endknoten gleich sind.
- *Färbungsproblem*: Man entscheide zu vorgegebener Zahl $k \in \mathbb{N}$, ob es eine Knotenmarkierung $\mu : V \rightarrow \{1, 2, \dots, k\}$ so gibt, daß für alle $(v, w) \in E$ gilt: $\mu(v) \neq \mu(w)$.
- *Cliquenproblem*: Man entscheide für ungerichteten Graphen G zu gegebener Zahl $k \in \mathbb{N}$, ob es einen Teilgraphen G' von G mit k Knoten gibt, dessen Knoten alle paarweise durch Kanten verbunden sind.
- *Kürzeste Wege* : Gegeben sei ein kantenbewerteter (ungerichteter) Graph $G = (V, E, \gamma)$ mit $\gamma : E \rightarrow \mathbb{R}^+$.
Es sei $W = (v_0, v_1, \dots, v_m)$ ein Weg. Dann heißt

$$C(W) = \sum_{i=1}^m \gamma(\{v_{i-1}, v_i\})$$
die *Länge* oder das *Gewicht* des Weges W .
 - Suche zu $v, w \in V$ einen kürzesten Weg $W_{\min}$ von v nach w , d.h. $C(W_{\min}) \leq C(W)$ für alle Wege W von v nach w .
 - Suche zu $v \in V$ und zu jedem $w \in V$ einen kürzesten Weg von v nach w .
- *Längste Wege*: Man finde zu zwei Knoten $v, w \in V$ einen *längsten Weg* von v nach w (evtl. bzgl. Kantenmarkierung).
- *Matching-Problem*: Eine Teilmenge $M \subseteq E$ der Kanten eines Graphen $G = (V, E)$ heißt *Matching*, wenn jeder Knoten von V zu höchstens einer Kante aus M gehört. Problem: finde ein *maximales Matching*, d.h. die Teilmenge(n) M mit größtmöglicher Kantenzahl, die die geforderte Eigenschaft erfüllt.

Unter den aufgezählten Problemen sind viele bekannte *NP-vollständige Probleme* (siehe Abschnitt 6.1), z.B. das Cliquesproblem, das Färbungsproblem, die Hamilton-Eigenschaftsprüfung und die Bestimmung längster Wege.

Darstellung von Graphen

Es gibt verschiedene Verfahren, um einen (un)gerichteten Graphen $G = (V, E)$ mit $|V| = n$ und $|E| = m$ darzustellen und zu implementieren; z.B. die *Adjazenzmatrix*:

Sei $V = \{1, 2, \dots, n\}$. Setze

$$a_{ij} = \begin{cases} \text{TRUE}, & \text{falls } (i, j) \in E \\ \text{FALSE}, & \text{falls } (i, j) \notin E \end{cases}$$

Dann läßt sich G durch eine quadratische Matrix der Ordnung n (Elemente vom Typ BOOLEAN) darstellen.

Bei großen Werten von n belegt die Adjazenzmatrix viel Speicher, obwohl sie meist nur sehr spärlich besetzt ist. In solchen Fällen bevorzugt man *Listendarstellungen*, die jeweils relevante Knoten und/oder Kanten verknüpfen.

Starke Zusammenhangskomponenten

An einem Beispiel wollen wir uns ansehen, wie graphentheoretische Probleme gelöst werden können. Zur Vorbereitung des Algorithmus zur Bestimmung starker Zusammenhangskomponenten (ZHK) folgen einige Begriffsbildungen und Aussagen.

1. Es sei $G = (V, E)$ ein gerichteter Graph.
 $H(G) = (V, E')$ heißt *transitive Hülle* von G : $\Leftrightarrow$
 $E' = \{(v, w) \mid v, w \in V \text{ und es existiert eine Kantenfolge von } v \text{ nach } w\}$.
2. Sei $G = (V_G, E_G)$ ein gerichteter Graph, $u, v \in V_G$.
 $u \sim v$ (*wegäquivalent*): $\Leftrightarrow$
es gibt eine Kantenfolge von u nach v und eine von v nach u .
3. K ist eine Äquivalenzrelation auf V , also reflexiv, symmetrisch und transitiv.
 $K[u]$ bezeichne die *Äquivalenzklasse* von u , d.h. $K[u] = \{v \in V \mid u \sim v\}$.
4. Sei $G = (V_G, E_G)$ ein gerichteter Graph.
 - (i) $\forall v \in V_G : G' = (K[v], E_G \cap (K[v] \times K[v]))$
ist eine starke Zusammenhangskomponente von G .
 - (ii) $G' = (V_{G'}, E_{G'})$ starke Zusammenhangskomponente von G
 $\Rightarrow \forall v \in V_{G'} : V_{G'} = K[v]$.

(iii) Es seien $K_1, K_2, \dots, K_r$ die Äquivalenzklassen bzgl. $\sim$.

Dann gilt $V_G = \bigcup_{i=1}^r K_i$ und für $i \neq j$: $K_i \cap K_j = \emptyset$.

Beispiel:

Man betrachte den Graphen $G = (V, E)$ mit $V = \{1, 2, 3, 4, 5, 6, 7\}$ und $E = \{(1,2), (1,4), (2,3), (3,1), (3,4), (3,6), (4,5), (5,6), (5,7), (7,5)\}$.

Die Äquivalenzklassen lauten dann:

$K_1 = \{1, 2, 3\}$, $K_2 = \{5, 7\}$, $K_3 = \{4\}$, $K_4 = \{6\}$.

Offensichtlich können diese Äquivalenzklassen angeordnet werden: Man kann im Graphen von Knoten aus K_1 zu Knoten in K_3 kommen, aber nicht umgekehrt. Wir notieren dies im Punkt 5(ii) durch $K_3 \leq K_1$.

5. Sei $G = (V_G, E_G)$ ein gerichteter Graph.

(i) $K(G) := \{K_1, K_2, \dots, K_r\}$. K_i wie oben eingeführt.

(ii) $K_i \leq K_j : \Leftrightarrow \exists u \in K_j \exists v \in K_i : \text{es existiert ein Weg von } u \text{ nach } v$.

“ $\leq$ ” ist eine Halbordnung auf $K(G)$, denn “ $\leq$ ” ist antisymmetrisch: Da die K_i jeweils starke Zusammenhangskomponente sind, kann kein Weg zurückführen.

6. Es existiert mindestens eine starke Zusammenhangskomponente von G , aus der keine Kante in andere Komponenten führt, nämlich ein K_i aus $K(G)$, das bzgl. “ $\leq$ ” minimal ist.

7. Es sei $G = (V, E)$ und $K(G) = \{K_1, K_2, \dots, K_r\}$.

Bilde $G_i = (V \setminus K_i, E \cap ((V \setminus K_i) \times (V \setminus K_i)))$, also den Graphen ohne K_i .

Dann gilt: $K(G_i) = K(G) \setminus \{K_i\}$.

Algorithmus zur Bestimmung aller starken Zusammenhangskomponenten

Wir wollen nach den Vorbereitungen des letzten Abschnitts die starken Zusammenhangskomponenten (ZHK) in einem gerichteten Graph bestimmen.

Grundsätzliches Vorgehen (nach Punkten 6. und 7. oben):

(1) Ermittle eine starke ZHK, aus der keine Kanten herausführen. Sei diese K .

(2) Bilde $(V \setminus K, E \cap ((V \setminus K) \times (V \setminus K)))$, d.h. lösche K und alle inzidenten Kanten, und setze das Verfahren mit diesem Graphen fort.

Wir werden jeden Knoten zunächst als eigene ZHK auffassen, alle seine Nachfolger untersuchen usw., bis sich ein Zyklus ergibt; alle Knoten auf dem Zyklus müssen in der gleichen ZHK liegen, sie werden daher zu einer Komponente zusammengefaßt. Indem man die bereits untersuchten, aber noch nicht zu einer ZHK ausgelagerten

Knoten auf einem *Stack* (siehe Abschnitt 3.2.4.2) ablegt, kann man sich die in Schritt (1) gewonnenen Erkenntnisse merken und in Schritt (2) direkt weiterverwenden (siehe Algorithmus).

Sei $G = (V, E)$ ein gerichteter Graph.

Im Algorithmus werden die Knoten und die Kanten von G *gefärbt* ("markiert"), und zwar bedeuten:

Knoten k ist blau gefärbt: Knoten k wurde noch nicht untersucht;
Knoten k ist rot gefärbt: k gehört bereits zu einer erkannten ZHK;
Knoten k ist grün gefärbt: k wird noch untersucht und gehört vorläufig zu einer aufzubauenden ZHK;
Kante e ist blau gefärbt: e wurde noch nicht untersucht;
Kante e ist rot gefärbt: e wurde bereits untersucht und wird nicht mehr betrachtet.

Sei $V = V_b \cup V_r \cup V_g$ (die Vereinigung ist disjunkt, V_b enthält die blauen, V_r die roten, V_g die grünen Knoten),
 $E = E_b \cup E_r$ (die Vereinigung ist disjunkt, E_b enthält die blauen, E_r die roten Kanten).

Die grünen Knoten werden auf einem Stack abgelegt. Die Nachfolgeknoten des jeweils zuletzt im Stack abgelegten Knotens werden im nächsten Schritt untersucht.

Beispiel

Die Knoten sind durch B (für blau), R (für rot) und G (für grün), die Kanten durch b (für blau) und r (für rot) gekennzeichnet.

Abb. 6.6 Zustand A von Graph und Stack

Der in Abb. 6.6 dargestellte Zustand A möge bereits vorliegen: der rechte, rot gefärbte Teil mit den Knoten 4, 5 und 6 ist bereits als ZHK erkannt, die Knoten wurden in der Reihenfolge 1, 2, 3, 4, 5, 6 untersucht und 1, 2, 3 liegen bereits auf dem Stack. Der Stack wird hier als lineare Liste dargestellt, in der die Schreib- und Leseoperationen auf dem gleichen Ende operieren.

Als nächstes sind die Nachfolger von 3 (= oberster Knoten im Stack) zu untersuchen, zu denen eine blau gefärbte Kante führt (hier: nur noch Knoten 7). Knoten 7 wird grün gefärbt und auf den Stack gebracht; Kante $3 \rightarrow 7$ ist rot zu färben. Nun wird mit Knoten 7 genauso verfahren. Ein Nachfolgerknoten, zu dem eine blau markierte Kante führt, ist der Knoten 6. Die Kante $7 \rightarrow 6$ wird rot gefärbt. Da aber 6 bereits zur ZHK gehört, wird dieser Knoten nicht weiter betrachtet. Der resultierende Zustand B von Graph und Stack ist in Abb. 6.7 repräsentiert.

Abb. 6.7 Zustand B von Graph und Stack

Als nächstes ist Kante $7 \rightarrow 3$ zu untersuchen und rot zu färben; man stellt fest, daß die Kante auf einen grünen (d.h. besuchten) Knoten führt, also ist ein Zyklus gefunden worden.

Alle Zyklus-Elemente (vom Top-Element des Stacks bis zum Zielknoten der Kante) gehören zur gleichen Komponente und werden im Stack zusammengefaßt (siehe Zustand C von Graph und Stack in Abb. 6.8).

Von den zusammengefaßten Knoten auf der obersten Keller-Ebene gehen nur noch rote Kanten aus, d.h. es liegt eine vollständige ZHK vor; deren Knoten sind rot zu färben und das oberste Stack-Element ist zu löschen. Daraus resultiert der in Abb. 6.9 dargestellte Zustand D von Graph und Stack.

Abb. 6.8 Zustand C von Graph und Stack

Abb. 6.9 Zustand D von Graph und Stack

Nun folgt mit Knoten 8 der Nachfolgeknoten von 2 auf den Stack usw.

Algorithmusentwurf

(* Programm P6.2 *)

```

 $E_b := E; E_r := \emptyset;$ 
 $V_b := V; V_r := \emptyset; V_g := \emptyset;$           (* Initialisierung *)
WHILE  $V_b \neq \emptyset$  DO  (* an dieser Stelle ist der Stack leer *)
    waehle irgendeinen Knoten  $v \in V_b$  aus;
     $V_b := V_b \setminus \{v\}$ ; faerbe  $v$  gruen, d.h.  $V_g := V_g \cup \{v\}$ ;
  
```

```

initialisiere den Stack mit v, d.h. Push (v);
WHILE stack  $\neq$  empty (entspricht  $V_G \neq \emptyset$ ) DO
  v := ein Knoten aus dem obersten Stack-Element;
  WHILE  $\exists e = v \rightarrow w$  : e ist blaue Kante DO
    faerbe e rot, d.h.  $E_b := E_b \setminus \{e\}$ ;  $E_r := E_r \cup \{e\}$ ;
    CASE Farbe von w OF
      | rot : (* nichts tun *)
      | gruen : (* Zyklus gefunden *)
        fasse alle Elemente des Stacks von oben bis zum
        Element, in dem w liegt, zum neuen obersten
        Element zusammen;
      | blau : faerbe w gruen;
         $V_b := V_b \setminus \{w\}$ ;  $V_g := V_g \cup \{v\}$ ;
        Push (w);
    END (* CASE *);
  END (* WHILE *);
  entferne oberstes Stack-Element als neue Komponente und
  faerbe alle seine Knoten rot; aktualisiere die Mengen  $V_g$ 
  und  $V_r$  entsprechend.
END (* WHILE *);
END (* WHILE *);

```

6.2.2 Bäume

Häufig sind Graphen wegen möglicher Zyklen für effiziente Algorithmen ungeeignete Datenstrukturen, da der Zyklentest zeitaufwendig ist und zusätzlichen Speicher für die Datenverwaltung erfordert. Außerdem lassen sich durch zyklensfreie Graphen die in der Informatik häufig anzutreffenden *Hierarchie*-Beziehungen und *konstruktive* Methoden (z.B. Erzeugungs- und Berechnungsverfahren) adäquat beschreiben. Deshalb werden als spezielle Graphen die sogenannten Bäume eingeführt, die *zyklensfrei* sind und einen eindeutigen "Einstiegspunkt" (die *Wurzel*) besitzen.

Nachfolgend werden Bäume nur als Spezialfälle gerichteter Graphen eingeführt. Es kann aber auch sinnvoll sein, "ungerichtete" Bäume zu betrachten.

1. Ein gerichteter Graph $G = (V_G, E_G)$ heißt *Baum*, wenn er keine Zyklen enthält und wenn es einen Knoten $r \in V_G$ gibt, so daß für alle $v \in V_G$ genau ein Weg von r nach v existiert.
 r heißt die *Wurzel* von G .
2. Ein Graph heißt *Wald*, wenn seine Komponenten Bäume sind.
Ein geordnetes m -Tupel von Bäumen $(T_1, T_2, \dots, T_m)$ heißt *geordneter Wald*.
3. Jeder Baum T läßt sich in der Form $(r; T_1, \dots, T_m)$ schreiben, wobei r die Wurzel von T ist und $T_1, \dots, T_m$ die an r "hängenden" *Teilbäume* von T sind.

$T = (r, T_1, \dots, T_m)$ heißt *geordnet*, wenn eine Anordnung $T_1 < T_2 < \dots < T_m$ vorgegeben ist, und wenn für $i \in \{1, \dots, m\}$ die T_i selbst geordnet sind.

Im folgenden sei $T = (V, E)$ ein Baum.

4. Unter dem *Grad* $d_T(v)$ eines Knotens v versteht man die Anzahl der von v ausgehenden Kanten.
5. T heißt *k-beschränkt* : $\Leftrightarrow \forall v \in V: d_T(v) \leq k$.
6. Ein 2-beschränkter, geordneter Baum heißt *Binärbaum*.
7. T heißt *k-gleichverzweigt* (oder auch: *vollständiger k-Baum*) : $\Leftrightarrow$
 $\forall v \in V: d_T(v) = k$ oder $d_T(v) = 0$.
8. $v \in V$ heißt *Blatt*: $\Leftrightarrow d_T(v) = 0$.
 $v \in V$ heißt *innerer Knoten* : $\Leftrightarrow d_T(v) \neq 0$.
9. Unter der *Höhe* lev eines Knotens v versteht man den um 1 erhöhten Abstand zur Wurzel r , d.h. $lev(v) = 1$, falls v die Wurzel ist, bzw. $lev(u) + 1$, falls u Vater von v ist.
10. $h(T) := \max \{lev(v) \mid v \in V\}$ heißt *Höhe eines Baumes*.
11. Ein Baum heißt *ausgeglichen* : $\Leftrightarrow$
für alle Blätter $v \in V$ gilt : $lev(v) = h(T)$ oder $lev(v) = h(T) - 1$.
Ein Baum heißt *vollständig ausgeglichen* : $\Leftrightarrow$
für alle inneren Knoten $v \in V$ gilt: die Anzahl der Knoten in den Teilbäumen von v unterscheidet sich höchstens um 1 und
für alle Blätter $v \in V$ gilt: $lev(v) = h(T)$ oder $lev(v) = h(T) - 1$.
12. T besitzt genau $|V| - 1$ Kanten.
13. Jeder Knoten v , der nicht Wurzel ist, besitzt genau einen Vorgänger u , den sog. *Vater(knoten)*, von dem aus eine gerichtete Kante auf v verweist. v heißt dann *Sohn(knoten)* von u . Haben v_1 und v_2 den gleichen Vater, so ist v_1 *Bruder(knoten)* von v_2 und umgekehrt.

Beispiele (τ ist jeweils die Wurzel)

Abb. 6.10 2 - beschränkt, 2 - gleichverzweigt

Jeder Knoten ist mit seiner Höhe markiert, die Höhe des Baumes ist 3, und an den Knoten stehen die jeweiligen Höhen der Knoten.

Abb. 6.11 2 - beschränkt, aber nicht 2 - gleichverzweigt

Abb. 6.12 ausgeglichen, 3 - beschränkt, aber nicht 3 - gleichverzweigt

Abb. 6.13 2 - beschränkt, 2 - gleichverzweigt, vollständig ausgeglichen

Darstellung von k-beschränkten Bäumen

Eine für k-beschränkte Bäume mit wenigen Knoten und/oder kleinem k durchaus adäquate Darstellung ist die folgende:

```

TYPE   KPtr = POINTER TO Ktree;
       Ktree = RECORD
           key : INTEGER;
           t1  : KPtr;
           ...
           tk  : KPtr;
       END (* RECORD *);

```

Der Nachteil bei $k \gg 2$ oder vielen Knoten und nicht vollständig ausgeglichener Struktur ist offensichtlich: die *maximale* Anzahl von Söhnen *eines* Knoten bestimmt die Anzahl der Zeiger für alle Knoten und impliziert daher einen möglicherweise unangemessen großen Speicherplatz wegen vieler NIL-Zeiger.

Die übliche, effiziente Darstellung durch *Binärbäume* beruht auf der Beobachtung, daß sich ein (geordneter) Baum durch Wurzel und eine Menge (Liste) disjunkter Unterbäume eindeutig beschreiben läßt:

$T = (r, T_{r_1}, T_{r_2}, \dots, T_{r_k})$, wobei T_{r_i} jeweils der Unterbaum mit der Wurzel r_i ist.

Dies legt folgende Struktur des binären Baumes nahe: linker Verweis auf einen Sohn, rechter Verweis auf einen Bruder (des nicht-binären Baums):

```

TYPE   BPtr = POINTER TO B;
       B    = RECORD
           node : inhalt;
           left  : BPtr;
           right : BPtr;
       END;

```

Ein Beispiel zeigt Abb. 6.14, wobei im oberen Teil ein 3-beschränkter Baum steht, der in eine normierte Binärbaum-Darstellung transformiert wird.

Abb. 6.14 Transformation eines allgemeinen Baums in einen Binärbaum

Traversierungs-Algorithmen

Eine häufig gewünschte komplexe Operation auf binären Bäumen ist ein “Durchlaufen” (*Traversieren*) in der Art, daß jeder Knoten genau einmal besucht wird. Man unterscheidet dabei *Inorder*-, *Preorder*- und *Postorder*-Durchlauf.

Die Eingabe ist jeweils ein Binärbaum, die Ausgabe die

- Inorder (erst linker Sohn, dann Vater, dann rechter Sohn)- bzw.
- Preorder (erst Vater, dann linker Sohn, dann rechter Sohn)- bzw.
- Postorder (erst linker Sohn, dann rechter Sohn, dann Vater)-

Reihenfolge seiner Knoteninhalte.

Die nachfolgenden Prozeduren zeigen die verschiedenen Durchlaufarten. Wie man sieht, ist nur die "Behandlung des Vaters" (durch eine "Action" genannte Operation) unterschiedlich eingeordnet.

Inorder - Durchlauf

```

PROCEDURE Inorder (ptr : NodePtr);          (* Programm P6.3a *)
(* Infix - Notation *)
BEGIN (* Inorder *)
  IF ptr # NIL THEN
    Inorder(ptr^.left);
    Action(ptr); (* Beliebige Operation auf ptr - Element *)
    Inorder(ptr^.right);
  END (* IF *);
END Inorder;

```

Preorder - Durchlauf

```

PROCEDURE Preorder (ptr : NodePtr);          (* Programm P6.3b *)
(* Prefix - Notation *)
BEGIN (* Preorder *)
  IF ptr # NIL THEN
    Action(ptr); (* Beliebige Operation auf ptr - Element *)
    Preorder(ptr^.left);
    Preorder(ptr^.right);
  END (* if *);
END Preorder;

```

Postorder - Durchlauf

```

PROCEDURE Postorder (ptr : NodePtr);          (* Programm P6.3c *)
(* Postfix - Notation *)
BEGIN (* Postorder *)
  IF ptr # NIL THEN
    Postorder(ptr^.left);
    Postorder(ptr^.right);
    Action(ptr); (* Beliebige Operation auf ptr - Element *)
  END (* IF *);
END Postorder;

```

Verschiedene Durchläufe erzeugen i.a. (nur zu Listen entartete Bäume bilden eine Ausnahme) natürlich auch unterschiedliche Ergebnisse. Für das Beispiel in Abb. 6.15 erhält man folgende Ergebnisse:

```

Inorder:    D B E A H F I C G
Preorder:   A B D E C F H I G
Postorder:  D E B H I F G C A

```

Abb. 6.15 Beispiel eines Binärbaumes

Für den Inorder-Durchlauf (Programm P6.3a) wollen wir nachfolgend eine nicht-rekursive Implementierung angeben.

```

PROCEDURE Inorder (q : BPtr);                                (* Programm P6.3d *)
(* Stack s ist eine globale Variable.*)
VAR p : BPtr;
BEGIN
  p := q;
  REPEAT
    WHILE p ≠ NIL DO
      Push (p, s);
      p := p^.left;
    END (* WHILE *);
    IF NOT Empty (s) THEN
      p := Top(s);
      Pop (s);
      WriteInt (p^.inhalt);  WriteLn;
      p := p^.right;
    END (* IF *);
  UNTIL Empty(s) AND (p = NIL);
END Inorder;

```

Die Analyse des Algorithmus ergibt, daß die maximale Länge des Stacks der Höhe des Baumes entspricht. Die n Knoten eines Baumes werden je einmal auf den Stack gelegt und wieder entfernt.

Auf die Abfrage $p = \text{NIL}$ stößt man $(n + 1)$ -mal, so daß sich wie für die rekursive Prozedur als asymptotische Zeitkomplexität $O(n)$ ergibt. Die kompliziertere Lösung bringt also kein Vorteile.

Fädelung

In einem binären Baum mit n Knoten sind $n+1$ Zeiger auf NIL gesetzt. Diesen Speicherplatz kann man nutzen, um Durchläufe effizienter zu gestalten, indem man diese Felder zu Referenzen ("Fäden") auf die bezüglich der Durchlaufart ausgezeichneten Vorgänger- und Nachfolger-Knoten nutzt. Ein solches Vorgehen nennt man *Fädelung*.

Um die "echten" Zeiger auf Söhne von den "unechten" Fäden zu bezüglich der Durchlaufart vorhergehenden oder nachfolgenden Knoten zu unterscheiden, muß für jeden der beiden Zeiger ein korrespondierendes Bit entsprechend gesetzt sein.

Ein *Beispiel* einer Inorder-Fädelung zeigt Abb. 6.16. Durchgezogene Kanten sind Zeiger auf Söhne, gestrichelte Kanten sind Fäden zu Inorder-Vorgänger und/oder-Nachfolger. Der Inorder-Vorgänger von D zeigt konsequenterweise auf NIL, ebenso der Inorder-Nachfolger von I.

Abb. 6.16 Inorder-Fädelung

Als Datenstruktur benutzen wir

```

TYPE   KPtr   = POINTER TO Knoten;
       Knoten = RECORD
           inhalt : M;
           lt, rt : KPtr;
           lbit, rbit : BOOLEAN;
       END;
```

Mit lbit, rbit unterscheiden wir, ob das jeweilige Zeigerfeld einen normalen Zeiger (TRUE) oder einen Faden (FALSE) enthält.

Die nachfolgende Tabelle stellt den ungefädelt und den gefädelt Fall für den Pre-, In- und Postorder-Durchlauf gegenüber, wobei

$x\text{-pred}(p) =$ Zeiger auf Vorgänger von $p^\wedge$ bei x -Durchlauf bzw.

$x\text{-suc}(p) =$ Zeiger auf Nachfolger von $p^\wedge$ bei x -Durchlauf, $x \in \{\text{pre}, \text{in}, \text{post}\}$.

ungefädelt	gefädelt
$p^\wedge.\text{lt} = \text{NIL}$	$p^\wedge.\text{lbit} \wedge p^\wedge.\text{lt} = x\text{-pred}(p)$
$p^\wedge.\text{lt} = g$	$p^\wedge.\text{lbit} \wedge p^\wedge.\text{lt} = g$
$p^\wedge.\text{rt} = \text{NIL}$	$p^\wedge.\text{rbit} \wedge p^\wedge.\text{rt} = x\text{-suc}(p)$
$p^\wedge.\text{rt} = g$	$p^\wedge.\text{rbit} \wedge p^\wedge.\text{rt} = g$

Algorithmus (Auffinden des Inorder-Nachfolgers in einem gefädelt Baum):

```

VAR q : KPtr;                                (* Programm P6.4 *)
BEGIN
  q := p^.rt;
  IF p^.rbit = 1 THEN
    WHILE q^.lbit = 1 DO
      q := q^.lt;
    END (* WHILE *);
  END (* IF *);
END;
```

Weitere Darstellungsformen von Bäumen

Auch die vorgestellte Binärdarstellung von Bäumen bzw. Wäldern benötigt noch sehr viel überflüssigen Speicherplatz, denn bei Verzicht auf Fädelungskonzepte gibt es bei n Knoten mit $2n$ Zeigern $n+1$ NIL-Zeiger. Eine naheliegende Idee ist deshalb, sich nur die Repräsentierung einer *Traversierung* des Baumes zu merken, um darauf zu operieren bzw. bei Bedarf den Baum zu (re-)konstruieren. Das Problem ist, daß *nur* aus der Darstellung z.B. eines Inorder-, Preorder- oder Postorder-Durchlaufs der Baum *nicht eindeutig* rekonstruierbar ist.

Eine Möglichkeit zur Verbesserung der Grundidee und zur Sicherung der Eindeutigkeit ist die, neben dem Traversierungsprotokoll noch *Zusatzinformationen* zu notieren. Nachfolgend stellen wir vier solche Darstellungsarten vor.

Postorder-Grad-Darstellung

Sei W ein Wald und B_W der zu W gehörende Binärbaum. Dann gilt: Der Postorder-Durchlauf von B_W und die Angabe des Grades zu jedem Knoten im Wald W genügen, um W eindeutig zu rekonstruieren. Für diese Darstellungsart wird der Postorder-Durchlauf sinngemäß auf Wälder erweitert.

Abb. 6.17 zeigt als Beispiel einen Wald aus 3 Bäumen.

Abb. 6.17 Wald mit 3 Bäumen

Der Postorder-Durchlauf von B_W ergibt folgende Knoten-Reihenfolge:
CEDBFAJIHLOMPKNKGRQ.

Die nachfolgende Tabelle gibt zu jedem Knoten seinen Grad im Wald W an:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Knoten	C	E	D	B	F	A	J	I	H	L	O	M	P	N	K	G	R	Q
Grad	0	0	1	2	0	2	0	1	1	0	0	1	0	1	3	2	0	1

Family-order-sequentielle Darstellung

Die Family-order-sequentielle Darstellung erinnert an einen "Breitendurchlauf" eines Baumes. Es werden stets Brüder unmittelbar nebeneinander gesetzt. Das Ende der Brüder wird durch "•" markiert. Die Verweise zeigen auf die am weitesten links stehenden Söhne.

```

ARRAY [1..N] OF RECORD
    inhalt : T;
    Bruderende : BOOLEAN;
    Sohn : 0..N;
END;
```

Eine lineare Darstellung (ebenfalls für das Beispiel aus Abb. 6.17) zeigt Abb. 6.18.

Abb. 6.18 Family-order-sequentielle Darstellung

Diese Darstellung eignet sich ebenfalls gut für Wälder. Man hat Freiheiten, in welcher Reihenfolge man die Unterbäume der Brüder notiert.

Kombinierte Preorder-Postorder-Darstellung

Die kombinierte Preorder-Postorder-Darstellung basiert auf der Angabe von Preorder- und Postorder-Durchlauf des zugehörigen Binärbaums:

```
ARRAY [1..N] OF RECORD pre, post : T END;
```

Hier wird vorausgesetzt, daß die Markierung den Knoten eindeutig bestimmt. Abb. 6.19 zeigt ein Beispiel eines Baumes und dessen Pre- und Postorder-Traversierung.

Abb. 6.19 Kombinierte Preorder-Postorder-Darstellung

Preorder-sequentielle Darstellung

Die Preorder-sequentielle Darstellung beruht auf der Idee, daß sich die übliche Darstellung eines Binärbaums mit einem Sohn- und einem Bruder-Zeiger komprimieren läßt, indem man als Sohn stets den nächsten Knoten in der Reihenfolge nimmt und "kein Sohn" (= Blatt) durch einen Boolean-Wert darstellt:

```

ARRAY [1..N] OF RECORD
    inhalt : T;
    Blatt : BOOLEAN;
    Bruder : 0..N;
END;

```

Dies setzt eine Umnummerierung der Knoten voraus. Wenn man das Beispiel aus Abb. 6.19 in Preorder-Reihenfolge durchläuft, ergibt sich in linearer Schreibweise (• = true) und als Array notiert die in Abb. 6.20 gezeigte Darstellung.

	Nummer ↓	inhalt	Blatt	Bruder
1	1	13		0
2	2	62		5
5	3	131	•	4
6	4	94	•	0
3	5	35	•	6
4	6	261		0
7	7	70		0
8	8	38	•	9
9	9	19	•	0

Abb. 6.20 Preorder-sequentielle Darstellung

Die 1. Spalte zeigt die alte, die 2. Spalte die neue Reihenfolge (gegeben durch Preorder). Auf diese Weise kann man auch einen Wald leicht darstellen, indem man die Wurzeln der einzelnen Bäume als Brüder auffaßt.

6.3 Suchen in gegebenen Datenstrukturen

Das Suchen nach bestimmten Daten ist eine häufige Operation in Programmen. Ein *Suchverfahren* ist ein Algorithmus, der - angewandt auf einen Datenbestand - für einen bestimmten *Schlüssel* (das *Suchargument*) die zugehörige Information, den eigentlich interessierenden *Inhalt*, ermittelt, falls ein Objekt mit diesem Schlüssel vorhanden ist. Im positiven Fall wird das *erste* "passende" Objekt als Ergebnis der erfolgreichen Suche geliefert, im negativen Fall wird die Suche erfolglos beendet.

Im Abschnitt 6.3 gehen wir aber davon aus, daß die Datenmenge, in der ein Element gesucht werden soll, gegeben ist und sich *nicht* mehr ändert. Das Thema "Suchen in sich ändernden Datenmengen" werden wir anschließend in Abschnitt 6.4 behandeln.

6.3.1 Suchen in Tabellen

Das Suchen von Daten in einem *Feld* (i.a. als *Array* repräsentiert) ist eine häufige Operation in Programmen. Gegeben sei folgender Datentyp für ein Feld, dessen Elemente jeweils aus einem *Schlüssel* (im folgenden jeweils vom Typ CARDINAL) und dem eigentlichen Inhalt bestehen :

```

TYPE    FeldTyp =    RECORD
                        Schlüssel: SchlüsselTyp;
                        (* z.B. CARDINAL *)
                        Inhalt : ...
                    END (* RECORD *);
VAR      F          :    ARRAY [1..FLaenge] OF FeldTyp;
        Wert       :    SchlüsselTyp;

```

Das Ziel des Suchens ist, im Feld F einen Eintrag mit dem Schlüssel Wert zu finden. Wir benötigen also einen möglichst effizienten Algorithmus, der uns entweder den *Index* des (ersten) Elements ausgibt, dessen Schlüssel mit Wert übereinstimmt, oder uns mitteilt, daß dieses Element nicht vorhanden ist.

Den folgenden Algorithmen legen wir das oben definierte Feld F zugrunde. Da nicht immer das ganze Feld durchsucht werden muß (eventuell ist es ja nicht vollständig belegt), führen wir als Index für das letzte tatsächlich belegte Element von F eine Variable n ein, für die gilt: $0 \leq n \leq \text{FLaenge}$.

Lineares Suchen

Liegen keine näheren Informationen über die *Ordnung* (z.B. eine aufsteigende Sortierung) des Feldes vor, so bleibt uns nichts anderes übrig, als es *sequentiell* zu durchlaufen und Element für Element zu prüfen:

```

i:=1;
WHILE (i <= n) AND (F[i].Schluessel # Wert) DO INC (i) END;
gefunden := (i <= n);

```

Verifikation

Die oben informell gegebene *Nachbedingung* ist:

$$Q \equiv (\neg \text{gefunden} \wedge \forall j \ 1 \leq j \leq n : F[j].\text{Schluessel} \neq \text{Wert}) \vee \\ (\text{gefunden} \wedge (\exists i \ 1 \leq i \leq n \wedge \forall j \ 1 \leq j < i : \\ F[j].\text{Schluessel} \neq \text{Wert} \wedge F[i].\text{Schluessel} = \text{Wert}))$$

Als Situation nach dem Verlassen der Schleife ergibt sich:

$$Q' \equiv \text{wp}(\text{Gefunden} := (i \leq n), Q) \\ \equiv (i > n \wedge \forall j \ 1 \leq j \leq n : F[j].\text{Schluessel} \neq \text{Wert}) \\ \vee (\exists i \ 1 \leq i \leq n \wedge \forall j \ 1 \leq j < i : \\ F[j].\text{Schluessel} \neq \text{Wert} \wedge F[i].\text{Schluessel} = \text{Wert})$$

Q' ist impliziert durch das *Abbruchkriterium* (also die negierte Wiederholbedingung) und die folgende *Invariante*:

$$\text{INV} \equiv (1 \leq i \leq n+1) \wedge (\forall j \ 1 \leq j < i : F[j].\text{Schluessel} \neq \text{Wert})$$

Man kann zeigen, daß die Invariante nach der Initialisierung und nach jeder Iteration gilt.

Aufwandsabschätzung:

Sofern die gesuchte Zahl im Feld vorhanden ist, gibt es im ungünstigsten Fall $n-1$ Schleifendurchläufe, im Durchschnitt $(n-1)/2$. Wenn sie nicht vorkommt, so wird die Schleife n mal durchlaufen.

Die einzige Chance, den Algorithmus zu beschleunigen, liegt in der Vereinfachung der Wiederholbedingung. Durch das Zufügen eines *Hilfselementes* mit dem Wert des gesuchten Elementes am Schluß des Feldes stellen wir sicher, daß der Algorithmus auch ohne die Abfrage auf das Feldende immer terminiert (setzt $n < \text{FLaenge}$ voraus). Diese Fassung wird als Suchen mit *Wächter* ("Sentinel") bezeichnet:

```

i:=1;
F[n+1].Schluessel := Wert;
WHILE F[i].Schluessel # Wert DO
  INC (i);
END;
gefunden := (i <= n);

```

Die Anzahl der Schleifendurchläufe bleibt im Wesentlichen die gleiche wie oben, durch die Vereinfachung der Wiederholbedingung wird die Schleife aber schneller durchlaufen. Suchen mit Sentinel ist also, abgesehen vom dafür benötigten Speicherplatz, etwas effizienter. Im Sinne der Asymptotik bleiben wir natürlich beim linearen Aufwand.

Binäres Suchen

Wenn das zu durchsuchende Feld *sortiert* ist, läßt sich die Suche mit *logarithmischem* (ld n für n Feldelemente) statt linearem Aufwand realisieren, und zwar durch das Prinzip des "Teilen und Herrschen" (*divide and conquer*): man bestimmt ein Mittelelement, vergleicht es mit gesuchten Element und je nach Ergebnis sucht man in einer der Hälften nach der gleichen Methode weiter.

Konkreter: wenn das Element der Mitte das gesuchte ist, kann die Suche beendet werden. Andernfalls wird der Bereich links oder rechts der Mitte durchsucht, je nachdem ob das Mittelelement zu groß oder zu klein war. Existiert der betreffende Bereich nicht, liegt also das Mittelelement schon am Rande, so war die Suche erfolglos.

Rekursiver Algorithmus A:

```

(* Programm P6.5a *)
PROCEDURE Search (      (* in *)   Wert: CARDINAL;
                        (* out *)  VAR gefunden: BOOLEAN;
                               VAR Pos: INTEGER);

  PROCEDURE SearchRec (Links, Rechts: INTEGER);
  BEGIN (* SearchRec *)
    IF Links <= Rechts THEN (* weitersuchen *)
      Pos := (Links + Rechts) DIV 2;
      IF F[Pos].Schluessel > Wert THEN
        (* linken Bereich durchsuchen *)
        SearchRec (Links, Pos-1);
      ELSIF F[Pos].Schluessel < Wert THEN
        (* rechten Bereich durchsuchen*)
        SearchRec (Pos+1, Rechts);
      ELSE
        gefunden := TRUE;
      END (* IF *);
    ELSE
      gefunden := FALSE;
    END (* IF *);
  END SearchRec;

  BEGIN (* Search *)
    SearchRec (1, n);
  END Search;
  (* Aufruf mit voller Laenge *)

```

Effizienterer iterativer Algorithmus B:

```

gefunden := FALSE;
Links := 1; Rechts := n;
(* Programm P6.5b *)
WHILE (Links <= Rechts) AND NOT gefunden DO
  Pos := (Links + Rechts) DIV 2;
  IF (F[Pos].Schluessel > Wert) THEN
    Rechts := Pos-1;
    (* linken Bereich durchsuchen *)
  ELSIF F[Pos].Schluessel < Wert THEN
    Links := Pos+1;
    (* rechten Bereich durchsuchen*)
  END IF;
END WHILE;

```



```

ELSE
    gefunden := TRUE;
END (*IF*);
END (* WHILE *);

```

Sowohl in Algorithmus A als auch in Algorithmus B steht am Ende die Position des gesuchten Elements in Pos (falls es gefunden wurde).

Auch der Algorithmus B läßt sich noch durch Vereinfachungen der Wiederholbedingung und des Schleifenrumpfes *beschleunigen*. Anstatt bei jedem Durchlauf auf Gleichheit zu testen, lassen wir den Algorithmus bis zum Ende laufen, d.h. bis die linke die rechte Grenze erreicht hat. Ob dort das gesuchte Element steht, wird dann aber nicht mehr in der Schleife abgefragt, so daß dafür eine *Spezialbehandlung* notwendig ist.

Beschleunigter Algorithmus C:

```

(* Programm P6.5c *)
PROCEDURE Search (    (* in *)    Wert: CARDINAL;
                    (* out *)    VAR gefunden: BOOLEAN;
                               VAR Pos: INTEGER);
VAR Links, Rechts: INTEGER;
BEGIN (* Search *)
    Links := 1;    Rechts := n;
    WHILE Links < Rechts DO
        Pos := (Links + Rechts) DIV 2;
        IF F[Pos].Schluessel < Wert THEN
            Links := Pos + 1    (* rechten Bereich durchsuchen*)
        ELSE
            Rechts := Pos;      (* linken Bereich durchsuchen *)
        END (* IF *);
    END (* WHILE *);
    Pos := Rechts;    gefunden := (F[Pos] = Wert);
END Search;

```

Bemerkenswert ist, daß diese Variante, wie auch das lineare Suchen, stets das gesuchte Element mit dem *kleinsten* Index zurückgibt. Bei den Algorithmen A und B ist dies nicht der Fall.

Aufwandsabschätzung für binäres Suchen

Es soll zuerst der Algorithmus C analysiert werden. Betrachten wir zunächst den Fall, daß das Feld aus $n = 2^m$ Elementen besteht; mit $m = 10$ z.B. folgt $n = 1.024$. Mit jeder Iteration sinkt die Breite des verbleibenden Intervalls auf die Hälfte, es sind also bis zum Abbruch $m = \log_2 n$ Durchläufe nötig (statt $\log_2 n$ schreiben wir abkürzend auch $\lg n$, logarithmus dualis). Ist n keine Zweierpotenz, so ist d , die Zahl der Durchläufe, der auf- oder abgerundete Wert von $\lg n$, also im ungünstigeren Fall $d = \lceil \lg n \rceil$.

Beim *Algorithmus B* ist d etwas geringer, falls der gesuchte Schlüssel vorhanden ist. Hat das Feld $n = 2^m - 1$ Elemente, so sind im ungünstigsten Falle m Durchläufe nötig, bis der Schlüssel gefunden ist, im günstigsten Fall genügt einer. Da aber der ungünstige Fall bei 2^{m-1} Schlüsseln auftritt, ist der *Durchschnitt* über alle vorhandenen Schlüssel:

$$d = \frac{1}{n} \sum_{i=1}^n d_i = \frac{1}{2^m - 1} \sum_{i=1}^m i \cdot 2^{i-1}$$

Vernachlässigt man für große m die "-1" im Nenner, so bildet sich für d der *Grenzwert*, dem sich d von oben nähert (Summe hier in umgekehrter Folge):

$$\begin{aligned} \lim d &= \frac{m}{2} + \frac{m-1}{4} + \frac{m-2}{8} + \dots + \frac{1}{2^m} = \frac{m}{2} + \frac{m}{4} + \frac{m}{8} + \dots - \frac{1}{4} - \frac{2}{8} \\ &\quad - \frac{3}{16} - \dots = m-1. \end{aligned}$$

Die Zahl der - aufwendigeren - Iterationen ist also kaum geringer. Der Algorithmus C erweist sich damit gegenüber B als umso überlegener, je größer das zu durchsuchende Feld ist.

6.3.2 Suchen von Zeichenketten

Im letzten Abschnitt haben wir die Suche nach einem einzelnen Element in einem Feld betrachtet. Dagegen ist die Suche nach einem ganzen *String* (nachfolgend als "*Pattern*" bezeichnet) in einem Text bedeutend aufwendiger. Wir werden zunächst dafür einen naiven und dann zwei effizientere Algorithmen vorstellen.

Naiver Algorithmus

Bei diesem Verfahren wird das Pattern *zeichenweise* am Text entlanggeschoben und jeweils *von Anfang an* verglichen, bis das Ende des Pattern oder eine Unstimmigkeit zwischen korrespondierenden Elementen des Pattern und des zu durchsuchenden Textes erreicht ist.

Algorithmus:

```
(* Programm P6.6 *)
(* Alle skalaren Variablen sind vom Typ CARDINAL, die Strings
   Text und Pattern sind ab 0 bis Length-1 indiziert *)

TextPos := 0;   PatternPos := 0;
WHILE (PatternPos < PatternLength)
  AND (TextPos + PatternLength <= TextLength) DO
    PatternPos := 0;
```



```

WHILE (PatternPos < PatternLength) AND
      (Pattern [PatternPos] = Text [TextPos + PatternPos])
DO
  INC (PatternPos);
END (*WHILE*);
INC (TextPos);
END (*WHILE*);
IF (TextPos > 0) THEN DEC (TextPos); END;
(* falls PatternPos >= PatternLength, wurde Pattern gefunden,
   und TextPos gibt jetzt die Position im Text an *)

```

Unter der Annahme, daß Unstimmigkeiten jeweils schon sehr früh, also bei den ersten Vergleichen auftreten, ist der Algorithmus recht gut. Probleme gibt es aber, wenn die Unstimmigkeit erst gegen Ende des Patterns auftritt (z.B. die Suche des Pattern "*****/" in einem Text der Form "****...****", was jedesmal erst beim 7. Vergleich eine Unstimmigkeit ergibt, auf die nur die Verschiebung um *ein* Zeichen folgt). In diesem ungünstigsten Fall werden $(t - s + 1) \cdot s$ Vergleiche ausgeführt, wobei t die Länge des Texts, s die des Patterns ist.

Der Suchalgorithmus von Knuth-Morris-Pratt (KMP)

Die Idee des KMP-Algorithmus besteht darin, aus dem Pattern gewisse Informationen über den bereits verglichenen Text herzuleiten. Wenn z.B. (vgl. Abb. 6.21) beim 5. Zeichen des Patterns eine Unstimmigkeit auftritt, sind die bereits verglichenen vier Zeichen des Textes bekannt. Damit ist auch klar, ob eine Verschiebung um weniger als vier Zeichen sinnvoll ist oder ob - wie in der Abb. 6.21 - sofort um vier Positionen verschoben werden kann.

Abb. 6.21 Vollständiges Weiterschieben mit Neuanfang des Patternvergleichs

Treten am Anfang des Pattern Wiederholungen von Zeichenfolgen auf (siehe Abb. 6.22), dann ist die zulässige Verschiebung geringer. Nach Feststellung der Diskrepanz auf der Vergleichsposition (Textzeichen ist nicht "B") könnte der Text auch "GEGEGEBEN" lauten. Darum muß nach Verschiebung um zwei Plätze das 3. Zeichen des Patterns an derselben Stelle des Textes verglichen werden.

Abb. 6.22 Teilweises Weiterschieben und Rest-Patternvergleich

Die zulässigen Verschiebungen werden *vor* dem eigentlichen Suchvorgang aus dem Pattern hergeleitet und in einer *Tabelle* gespeichert (Prozedur *TabelleBauen* in der Prozedur KMP im Programm P6.7 am Ende des Abschnitts 6.3.2.). Die einzelnen Elemente der Tabelle geben jeweils an, an welche Stelle links der Ungleichheit das Pattern verschoben werden kann (siehe Abb. 6.23). Die Stelle im Text, an der verglichen werden muß, bleibt gleich oder wird um eine Position nach rechts verschoben, wenn die neue Position des Patternanfangs *rechts* von der Vergleichsposition steht.

-1	0	-1	0	2	0	0	0	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Verschiebungstabelle
(Gleich[])

- # Ungleichheit zwischen Text und Pattern
 - Neue Startposition des Patternanfangs bei Ungleichheit
 - ➔ Fortsetzung mit nächstem Textzeichen
 - ↓↑ Fortsetzung mit aktuellem Textzeichen
- Die Pfeile zeigen auf die für den jeweiligen Fall geltende neue Startposition des Pattern

Abb. 6.23 Tabellenbau bei KMP

Bedeutung der Werte in 'Gleich[]':

- Gleich[0] = -1 Ungleichheit bei erstem Pattern-Zeichen, weiter zum *nächsten* Textzeichen;
- Gleich[i] = -1 gleiches Zeichen wie am Anfang erwartet, aber nicht angetroffen, weiter zum *nächsten* Textzeichen mit Pattern-Anfang;
- Gleich[i] = 0 Teste am *aktuellen* Textzeichen den Pattern-Anfang neu (anfangs ohne Wiederholung);
- Gleich[i] = p > 0 Teste am *aktuellen* Textzeichen das Pattern ab Index p (Länge der Wiederholung).

Nach dem jetzt fälligen Vergleich "R" gegen "T" in der Abb. 6.23 wird das Pattern um 1 verschoben, und die Vergleichsposition im Text wandert zum "M".

Aufwandsabschätzung:

Der Aufwand bei KMP ist in extremen Fällen deutlich *geringer* als der des naiven Verfahrens. Die Anzahl der Vergleiche beträgt höchstens $2t - s$.

Begründung: Zwischen zwei Vergleichen wird entweder das Pattern geschoben (kann höchstens $t - s$ mal geschehen) oder die Vergleichsposition wandert nach rechts (höchstens $t - 1$ mal). Daher sind nicht mehr als $t - s + t - 1 + 1 = 2t - s$ Vergleiche möglich. In der Regel sind es kaum mehr als t .

Ein weiterer Vorteil des KMP-Algorithmus ist, daß der Textzeiger während des Suchens nicht zurückgesetzt werden muß. Dies ist besonders bei der Suche in sequentiellen *Dateien* nützlich.

Als *Nachteil* kann man den Aufbau (Komplexität: $O(s)$) und die Speicherung der Tabelle betrachten. Speichermäßig benötigt sie mindestens soviel Platz wie das

Pattern selbst; die Berechnung lohnt sich vor allem bei kurzen Patterns (und Texten) nicht.

Zusammenfassend läßt sich feststellen, daß der KMP-Algorithmus den ungünstigsten Fall erheblich entschärft.

Der Suchalgorithmus von Boyer-Moore (BM)

Wie schon beim KMP- wird auch beim BM-Algorithmus zuvor aus dem Pattern eine Tabelle entwickelt, die die Suche erheblich beschleunigt.

Anders als bei KMP wird hier aber

- das Pattern *von hinten nach vorne* (beginnend mit dem letzten Zeichen des Patterns) mit dem aktuellen Textfenster verglichen,
- bei einer Diskrepanz die notwendige Verschiebung des Patterns von dem Textzeichen abgeleitet, das mit dem *letzten* Zeichen des Patterns verglichen wurde. Das Pattern wird dann soweit verschoben, bis es an das betreffende Zeichen "paßt" oder erst rechts davon beginnt.

Die Verschiebung des Pattern wird in einer Tabelle (DELTA) verwaltet, in der zu jedem Zeichen des Zeichensatzes dessen *letzte Position* im Pattern oder, falls das Zeichen im Pattern nicht vorkommt, die Länge des Pattern gespeichert wird. Abb. 6.24 zeigt als Beispiel einen Suchvorgang mit dem BM-Algorithmus.

Abb. 6.24 Suchalgorithmus nach Boyer und Moore

Die Tabelle DELTA hat für das Pattern "GENO" folgenden Inhalt:

DELTA [A] = 4

...

DELTA [E] = 2

DELTA [F] = 4

DELTA [G] = 3

DELTA [H] = 4


```

...
DELTA [N] = 1
DELTA [O] = 4 (* da O am Ende *)
...

```

Erläuterungen (die Zahlen bezeichnen die Vergleiche in Abb. 6.24):

- 1,2 *Übereinstimmung*: Weiter von rechts nach links vergleichen.
- 3 *Diskrepanz*: Pattern wird um vier Plätze, also die Länge, geschoben, weil "O" außer auf der letzten Position nicht mehr im Pattern vorkommt. Vergleich ab "E" im Text.
- 4 *Diskrepanz*: Pattern wird um zwei Plätze geschoben, weil "E" zwei Plätze vor dem letzten Zeichen im Pattern vorkommt. Vergleich ab "O" im Text.
- 5-8 *Gefunden!*

Aufwandsabschätzung:

Der BM-Algorithmus, der hier in der vereinfachten Fassung von Wirth wiedergegeben ist, verbessert die Effizienz der Suche im durchschnittlichen Fall ganz erheblich (bis etwa t/s Vergleiche für ein nicht enthaltenes Pattern); in ungünstigen Fällen ist er nicht besser als der naive Algorithmus. Daher haben die Autoren auch eine *Kombination* mit dem KMP-Algorithmus vorgeschlagen. Ein Nachteil des BM-Algorithmus ist, daß der Text nicht sequentiell durchsucht wird (Rücksetzen des Textzeigers), so daß bei der Speicherung des Textes in sequentiellen Dateien Probleme entstehen.

Anwendungsbeispiel: Für (englische Teilwort-)Pattern mit Länge 5 in englischen Texten werden im statistischen Mittel ca. $t/4$ Vergleiche benötigt).

Die ausführlichen Beschreibungen und Analysen der KMP- und BM-Algorithmen finden sich in

- Knuth, Morris, Pratt: Fast pattern matching in strings; SIAM Journal of Comput. 6, 323-349
- Boyer, Moore: A fast string searching algorithm; Communications of the ACM 20, 762-772.

Implementierungen

Abschließend wollen wir noch die vollständige Implementierung der drei aufgeführten Suchalgorithmen angeben (aus Gründen der Übersichtlichkeit mit globalen Variablen). Um den Aufwand vergleichen zu können, sind die *Vergleiche* in eine spezielle Funktion Comp ausgelagert, wo jeweils als Nebenwirkung eine Zählvariable erhöht wird. Die Prozedur TabelleBauen zur Berechnung der Pattern-Verschiebung im KMP-Algorithmus wurde zwar nur vorbereitend erläutert, wird aber hier zur Vervollständigung der Implementierung angegeben.

```

MODULE Stringsearch;                                (* Programm P6.7 *)
FROM InOut IMPORT OpenInput, CloseInput, OpenOutput,
                  CloseOutput, Read, Write, WriteString,
                  WriteCard, WriteInt, WriteLn, Done;

CONST  LngStrMax = 100;
       LngTxtMax = 80;
       MINCHAR   = 0;
       MAXCHAR   = 127;
VAR    Txt  : ARRAY [0..LngTxtMax-1] OF CHAR;      (* Text *)
       Str  : ARRAY [0..LngStrMax-1] OF CHAR;      (* Suchstring/Pattern *)
       LngStr, LngTxt : INTEGER;

PROCEDURE GetText;
CONST  CR = 15C;
VAR    Zeichen : CHAR;
BEGIN (* GetText: Text einlesen *)
  WriteString ("Bitte den Namen des Textfiles eingeben:");
  WriteLn;
  OpenInput ("TEXT");  LngTxt:=0;  Read (Zeichen);
  WHILE Done AND (LngTxt < LngTxtMax) DO
    Txt[LngTxt] := Zeichen;  INC (LngTxt);
    IF Zeichen = CR THEN WriteLn
      ELSE Write (Zeichen);  END (* IF *);
    Read (Zeichen);
  END (* WHILE *);
  CloseInput; WriteLn;
  WriteInt (LngTxt,8); WriteString ('Zeichen gelesen');
  WriteLn;
END GetText;

PROCEDURE GetPattern;
VAR    Zeichen : CHAR;
BEGIN (* GetPattern *)
  WriteLn;  WriteString ("Bitte Suchstring eingeben:");
  LngStr := 0;
  LOOP (* Suchstring einlesen *)
    Read (Zeichen);
    IF Zeichen < "" THEN EXIT END;
    Write (Zeichen);  Str [LngStr] := Zeichen;
    INC (LngStr);
    IF LngStr >= LngStrMax THEN EXIT END;
  END (* LOOP *);
  WriteLn;
END GetPattern;

PROCEDURE Comp ( LogExp : BOOLEAN;
                 VAR Count : CARDINAL) : BOOLEAN;
(* zum Zaehlen der Vergleiche *)
BEGIN (* Comp *)
  INC (count);
  RETURN LogExp;
END Comp;

```



```

PROCEDURE Results (Header: ARRAY OF CHAR; Gefunden: BOOLEAN;
                  Position, AnzIndVgl, AnzZchVgl: CARDINAL);
BEGIN (* Results *)
  WriteString (Header);
  IF Gefunden THEN
    WriteString ("Gefunden bei Position");
    WriteInt (Position, 5);
    (* Die Zahl der ueberlesenen Zeichen wird ausgegeben *)
  ELSE
    WriteString ("Nicht gefunden      ");
  END (* IF *);
  WriteString (" Vergleiche Index/Zeichen");
  WriteCard (AnzIndVgl, 5);  WriteCard (AnzZchVgl, 5);
  WriteLn;
END Results;
(*****

PROCEDURE SimpleSearch;          (* Naiver Suchalgorithmus *)
VAR  Position, CheckPos  : INTEGER;
      AnzIndVgl, AnzZchVgl : CARDINAL;
BEGIN (* SimpleSearch *)
  AnzIndVgl := 0;  AnzZchVgl := 0;
  Position := 0;  CheckPos := 0;
  WHILE Comp (CheckPos < LngStr, AnzIndVgl)
                                (* d.h. nicht gefunden *)
    AND Comp (Position <= LngTxt - LngStr, AnzIndVgl) DO
                                (* d.h. Treffer ist noch moeglich *)
    CheckPos := 0;
    WHILE Comp (CheckPos < LngStr, AnzIndVgl)
      AND Comp (Txt [Position + CheckPos] =
                Str [CheckPos], AnzZchVgl) DO
      INC (CheckPos);
    END (* WHILE *);
    INC (Position);
  END (* WHILE *);
  IF Position > 0 THEN DEC (Position)  END;
  Results ("Ergebnis naiv:", CheckPos <= LngStr,
          Position, AnzIndVgl, AnzZchVgl);
END SimpleSearch;
(*****

PROCEDURE KMP;          (* Knuth-Morris-Pratt-Algorithmus *)
VAR  TextInd, StrInd : INTEGER;
      AnzIndVgl, AnzZchVgl : CARDINAL;
      (* Zaehler fuer Vergleiche *)
      Gleich : ARRAY [0..LngStrMfax-1] OF INTEGER;
      (* Tabelle fuer Verschieb. *)

PROCEDURE TabelleBauen;  (* Tabelle fuer KMP *)
VAR  StrInd,
      (* durchlauft String bis zum
        zweitletzten Zeichen *)
      VglInd: INTEGER;  (* Laenge d. linken Teilstrings, der
        mit Stringanfang uebereinstimmt. *)

```



```

BEGIN (* TabelleBauen *)
  Gleich [0] := -1;      (* bleibt konstant *)
  VglInd := 0;
  FOR StrInd := 1 TO LngStr-1 DO
    IF Str [StrInd] = Str [VglInd] THEN
      Gleich [StrInd] := Gleich [VglInd];
    ELSE
      Gleich [StrInd] := VglInd;
    END (* IF *);
    WHILE (VglInd >= 0)
      AND (Str [StrInd] <> Str [VglInd]) DO
      VglInd := Gleich [VglInd];
    END (* WHILE *);
    INC (VglInd);
  END (* FOR *);
END TabelleBauen;

BEGIN (* KMP *)
  TabelleBauen;
  AnzIndVgl := 0, AnzZchVgl := 0;
  TextInd := 0, StrInd := 0;
  WHILE Comp (StrInd < LngStr, AnzIndVgl) AND
    (* String nicht gefunden *)
    Comp (TextInd < LngTxt, AnzIndVgl) DO
    (* Text nicht abgearbeitet *)
    WHILE Comp (StrInd >= 0, AnzIndVgl) AND
      (* Vergleich noch notwendig *)
      Comp (Txt [TextInd] <> Str [StrInd], AnzZchVgl) DO
      StrInd := Gleich [StrInd];
      (* Verschiebung vornehmen *)
    END (* WHILE *);
    (* gleiche Zeichen gefunden und/oder StrInd = -1 *)
    INC (TextInd);      (* naechstes Zeichen *)
    INC (StrInd);       (* ...vergleichen *)
  END (* WHILE *);
  Results ("Ergebnis KMP: ", StrInd=LngStr,
    TextInd-LngStr, AnzIndVgl, AnzZchVgl);
End KMP;
(*****)

PROCEDURE BM;      (* Boyer-Moore-Algorithmus *)
VAR TxtInd, StrInd, VglInd : INTEGER;
    AnzIndVgl, AnzZchVgl : CARDINAL;
    Delta : ARRAY [MINCHAR .. MAXCHAR] OF INTEGER;
    (* Tabelle fuer Verschiebung *)

PROCEDURE TabelleBauen; (* Tabelle fuer BM-Algorithmus *)
VAR StrInd : INTEGER; (* durchlaeuft String rueckwaerts
    ab zweitletztem Zeichen *)
    Zeichen : CARDINAL; (* Ordnungszahl eines Zeichens*)
BEGIN (* TabelleBauen *)
  FOR Zeichen := MINCHAR TO MAXCHAR DO
    Delta [Zeichen] := LngStr;
  END (* FOR *);

```

```

    FOR strInd := 0 TO LngStr-2 DO
        Delta [Str [StrInd]] := LngStr - StrInd - 1;
    END (* FOR *);
END TabelleBauen;

BEGIN (* BM *)
    TabelleBauen;
    AnzIndVgl := 0;    AnzZchVgl := 0;
    StrInd := LngStr-1;    (* wegen WHILE-Bedingung, s. unten *)
    TxtInd := StrInd;
    WHILE Comp (StrInd >= 0, AnzIndVgl)
        AND Comp (TxtInd < LngTxt, AnzIndVgl) DO
        StrInd := LngStr-1;    VglInd := TxtInd;
    LOOP
        IF Comp (StrInd < 0, AnzIndVgl) THEN EXIT END;
                                                (* Gefunden *)
        IF Comp (Str[StrInd] # Txt[VglInd], AnzZchVgl) THEN
            TxtInd := TxtInd + Delta[ORD(Txt[TxtInd])];
                                                (* Verschieben *)
            EXIT;
        END (* IF *);
        DEC (StrInd);                                (* Rueckwaerts *)
        DEC (VglInd);                                (* vergleichen *)
    END (* LOOP *);
    END (* WHILE *);
    Results ("Ergebnis BM : ", StrInd < 0,
            TxtInd-LngStr + 1, AnzIndVgl, AnzZchVgl);
End BM;
(*****)

BEGIN (* Stringsearch *)
    GetText;
    LOOP
        GetPattern;
        IF Str [0] = '.' THEN EXIT END;
        SimpleSearch;    KMP;    BM;
    END (* LOOP *);
END Stringsearch.

```

Der nachfolgende - verkürzte - Ausdruck zeigt den Anfang eines *Testlaufs*:

Bitte den Namen des Textfiles eingeben: Testtext

*****...verkürzt...*****/***
80 Zeichen gelesen.

Bitte Suchstring eingeben: *

Ergebnis naiv: Gef. bei Position 0 Vergl. Index/Zeichen: 5 1

Ergebnis KMP: Gef. bei Position 0 Vergl. Index/Zeichen: 4 1

Ergebnis BM : Gef. bei Position 0 Vergl. Index/Zeichen: 5 1

Bitte Suchstring eingeben: /***

Ergebnis naiv: Gef. bei Position 76 Vergl. Index/Zeichen: 236 80

Ergebnis KMP: Gef. bei Position 76 Vergl. Index/Zeichen: 317 80

Ergebnis BM: Gef. bei Position 76 Vergl. Index/Zeichen: 449 297

Die vollständigen Ergebnisse sind in der folgenden Tabelle wiedergegeben (tatsächlich bestand der Text aus insgesamt 79 Zeichen '*', vor den drei letzten Sternen eingeschoben ein Slash; für jeden der drei Algorithmen (Naiv, KMP, BM) sind jeweils in der 1. Spalte die Anzahl der Index- und in der 2. Spalte die Anzahl der Zeichenvergleiche angegeben):

Pattern	Position	Naiv		KMP		BM	
*	0	5	1	4	1	5	1
/***	76	236	80	317	80	449	297
***/*	73	446	296	305	150	227	77
/***/*	-	230	76	318	80	225	77
	0	1	0	1	0	1	0
abc	-	236	78	322	80	80	26
*****/*	63	1026	896	295	140	207	77
*/**	-	308	154	319	157	159	79
/***/*	-	230	76	320	80	229	77

Wie man sieht, ist der naive Algorithmus in einem Großteil der Fälle gar nicht so viel schlechter. Dazu kommt, daß er sehr viel einfacher zu implementieren und zu warten ist. Es zeigt sich damit wieder, daß vor der Auswahl einer Algorithmusvariante die Situation in der Einsatzumgebung gründlich zu analysieren ist.

6.4 Datenorganisationen für effizientes Suchen

In Abschnitt 6.3 haben wir Suchverfahren für gegebene, nicht durch Einfügen und Löschen ihrer Elemente dynamisch wachsende oder schrumpfende Datenbestände behandelt.

Im Abschnitt 6.4 geben wir diese Beschränkung auf und stellen in 6.4.1 *Bäume* als Datenorganisation für einen effizienten Zugriff beim Suchen von Elementen vor. Die Elemente sind i.a. in den Knoten der Bäume selbst enthalten, doch kann man sich Bäume auch als reine "Suchstrukturen" vorstellen, die nur den Weg zu den zu suchenden Elementen weisen und nur einen Zeiger oder einen Index liefern, mit dem man dann schließlich das gesuchte Element findet.

Abschnitt 6.4.2 behandelt das *Hashing*. Im Gegensatz zu Bäumen, in denen das Suchen durch *Schlüsselvergleiche* erfolgt, wird beim Hashing aus dem *Suchargument* direkt ein Schlüssel berechnet. D.h. man versucht, durch eine Adressenberechnung das gesuchte Element direkt zu finden oder zumindest mit der Adresse ganz in der Nähe zu "landen".

6.4.1 Suchverfahren auf Bäumen

Das Suchen in *Listen* (ohne Sortierung) mit n Elementen dauert im schlechtesten Fall n Vergleiche, bis man das Element gefunden oder die erfolglose Suche festgestellt hat. Es liegt nahe, dies zu beschleunigen, indem man Bäume verwendet, deren Eigenschaften die Suche erleichtern. Solche Eigenschaften sind z.B. Zusicherungen, daß sich in einem Teilbaum nur Knoten mit einem eingeschränkten Wertebereich (z.B. alle "kleineren" oder "größeren" Knoten) befinden.

6.4.1.1 Binäre Suchbäume

Unter einem *binären Suchbaum* über einer linear geordneten Menge X verstehen wir einen geordneten, gerichteten, mit Elementen aus X knotenmarkierten *Binärbaum*, für dessen *Markierung* M in jedem Knoten gilt (wir schließen eine wertgleiche Markierung von Knoten aus!):

- falls k einen linken Nachfolger besitzt, dann ist $M(k') < M(k)$ für alle Knoten k' im linken Unterbaum von k , und
- falls k einen rechten Nachfolger k'' besitzt, dann ist $M(k) < M(k'')$ für alle Knoten k'' im rechten Unterbaum von k .

Der binäre Suchbaum ist also bezüglich der $<$ -Relation linear geordnet. Das *Suchen* eines Elementes in einem binären Suchbaum geschieht, indem beginnend mit der Wurzel der Wert x_1 des zu suchenden Elementes mit dem Wert x_2 des jeweils aktuell betrachteten Baumknotens k verglichen wird. Falls $x_1 = x_2$, so ist das Element gefunden, falls $x_1 < x_2$, betrachtet man die Wurzel des linken, falls $x_1 > x_2$ die Wurzel des rechten Teilbaums von k . Falls der entsprechende Teilbaum leer ist, so bleibt die Suche erfolglos.

Das *Einfügen* eines Elementes basiert auf der gleichen algorithmischen Idee wie beim Suchen mit folgenden Besonderheiten: Im Falle $x_1 = x_2$ kann das Element - da bereits vorhanden - nicht mehr eingefügt werden, beim Feststellen eines leeren Teilbaums ist die Einfügestelle gefunden und das Einfügen kann erfolgen.

Beispiel: Gegeben sei die Folge ganzer Zahlen 14, 23, -2, 12, -5, 36, 0, 28, 20.

Das sukzessive Aufbauen des binären Suchbaums für die gegebene Zahlenfolge durch fortlaufendes Einfügen verdeutlicht Abb. 6.25.

Ein *Inorder-Durchlauf* liefert dann diese Zahlenfolge in aufsteigender Sortierung: -5, -2, 0, 12, 14, 20, 23, 28, 36.

Das *Löschen* eines Elementes k in einem binären Suchbaum geschieht dadurch, daß k durch seinen Inorder-Nachfolger "ersetzt" wird.

- Fall 1: $k.left = NIL$, dann ersetze k durch rechten Teilbaum, d.h. ziehe die Wurzel k' des *rechten* Teilbaums hoch und ersetze k durch k' ;
- Fall 2: $k.right = NIL$, dann ersetze k durch linken Teilbaum, d.h. ziehe die Wurzel k'' des *linken* Teilbaums hoch und ersetze k durch k'' ;
- Fall 3: *sonst*, dann ersetze k entweder durch das kleinste Element im rechten Teilbaum (*Inorder-Nachfolger*) oder durch das größte Element im linken Teilbaum (*Inorder-Vorgänger*). Im MODULA-2-Programm P6.8 wurde die zweite Variante gewählt.

Die Suche nach dem Ersatzelement erfolgt in einer eingeschachtelten rekursiven Prozedur. Diese sucht vom rechten Sohn des zu löschenden Knotens nach links, bis ein Knoten keinen linken Sohn hat. Dieser ersetzt dann den zu löschenden Knoten und wird seinerseits durch die Wurzel seines rechten Unterbaums ersetzt.

Abb. 6.25 Sukzessives Aufbauen eines binären Suchbaums

In Programm P6.8 ist eine komplette Modula-2-Implementierung für das *Einfügen*, das *Suchen* und das *Löschen* von Elementen in einem binären Suchbaum angegeben.

Man stelle sich vor, daß der binäre Suchbaum in seinen Elementen neben den Schlüsseln die an sich interessanten Inhalte (für die der Suchbaum mit seinen Schlüsseln ja nur eine *Zugriffsstruktur* darstellt) enthält.

Zum Verständnis des Programms P6.8 hilft die Anschauung, daß damit eine Tabelle verwaltet wird, wobei der Schlüssel ein String und der Inhalt nur ein einfacher Real-Wert ist.

Schnittstelle:

```

DEFINITION MODULE BinSuchBaum;                                (* Programm P6.8 *)
(* ZWECK   : Tabellenverwaltung mit binaerem Suchbaum *)

TYPE Zeichenkette = ARRAY [0..20] OF CHAR;
   SchluesselTyp = RECORD
       String: Zeichenkette;
       Laenge: CARDINAL;
   END (* RECORD *);

   InhaltTyp    = REAL;  (* nur als Beispiel fuer einen
                           (i.a. umfangreichen) Typ *)

   ElementTyp   = RECORD
       Schluessel : SchluesselTyp;
       Inhalt     : InhaltTyp;
   END (* RECORD *);

PROCEDURE Einfuegen ( (* in *) Element : ElementTyp;
                     (* out *) VAR OK   : BOOLEAN);
(* Traegt Element in die Tabelle ein:
   OK = TRUE : Eintrag erfolgreich durchgefuehrt
   OK = FALSE: Tabelle ist voll o. Eintrag schon vorhanden *)

PROCEDURE Suchen ( (* in *) SuchSchluessel : SchluesselTyp;
                  (* out *) VAR Element   : ElementTyp;
                  (* out *) VAR OK        : BOOLEAN);
(* Findet Element mit Schluessel 'SuchSchluessel':
   OK=TRUE : Element gefunden.
   OK=FALSE: Element war nicht in der Liste. *)

PROCEDURE Loeschen ( (* in *) LoeschSchluessel : SchluesselTyp;
                    (* out *) VAR OK           : BOOLEAN);
(* Loescht Element mit Schluessel 'LoeschSchluessel':
   OK=TRUE : Element geloescht.
   OK=FALSE: Element war nicht in der Liste *)

END BinSuchBaum.
```

Implementierung:

```

IMPLEMENTATION MODULE BinSuchBaum;
(* ZWECK : Tabellenverwaltung mit binaerem Suchbaum *)

FROM Storage IMPORT ALLOCATE, DEALLOCATE;
```

```

TYPE ZweigTyp    = POINTER TO KnotenTyp;
   KnotenTyp    = RECORD
       Inhalt : ElementTyp;
       UBLinks, UBRechts : ZweigTyp;
   END (* RECORD *);
VAR Gesamtbaum : ZweigTyp;
TYPE VglResTyp = (Vor, Gleich, Nach);

(***** interne Prozedur *****)
PROCEDURE StringVgl (S1, S2: SchluesselTyp) : VglResTyp;
(* gibt an, ob S1 lexikographisch vor oder nach S2 kommt
   oder ob S1 gleich S2 ist. *)
VAR i, n : CARDINAL;

   PROCEDURE Minimum (A, B: CARDINAL): CARDINAL;
   BEGIN (* Minimum *)
       IF A < B THEN RETURN A ELSE RETURN B END;
   END Minimum;

   BEGIN (* StringVgl *)
       n := Minimum (S1.Laenge, S2.Laenge);
       i := 0;
       WHILE (i <= n) AND (S1.String[i] = S2.String[i]) DO
           INC (i);
       END (* WHILE *);
       IF i <= n THEN
           IF S1.String[i] < S2.String[i] THEN
               RETURN Vor;
           ELSE
               RETURN Nach;
           END (* IF *);
       ELSE
           IF n < S1.Laenge THEN
               RETURN Nach;
           ELSIF n < S2.Laenge THEN
               RETURN Vor;
           ELSE
               RETURN Gleich;
           END (* IF *);
       END (* IF *);
   END StringVgl;

(*****
PROCEDURE Einfuegen ( (* in *) Element : ElementTyp;
                      (* out *) VAR OK: BOOLEAN);

   PROCEDURE RekeEintragen (VAR Baum: ZweigTyp);
   BEGIN (* Eintragen *)
       IF Baum = NIL THEN (* neues Element im Binaerbaum
                           anhaengen *)
           ALLOCATE (Baum, SIZE (KnotenTyp));
           WITH Baum^ DO
               UBLinks := NIL; UBRechts := NIL; Inhalt := Element;
           END (* WITH *);
           OK := TRUE;

```



```

ELSE
  WITH Baum^ DO
    CASE StringVgl (Element.Schluessel, Inhalt.Schluessel) OF
      | Vor      : RekEintragen (UBLinks);
      | Nach     : RekEintragen (UBRechts);
      | Gleich   : OK := FALSE;
    END (* CASE *);
  END (* WITH *);
END (* IF *);
END RekEintragen;

BEGIN (* Einfuegen *)
  RekEintragen (Gesamtbaum);
END Einfuegen;

(*****)
PROCEDURE Suchen ( (* in *) SuchSchluessel : SchluesselTyp;
                  (* out *) VAR Element : ElementTyp;
                  (* out *) VAR OK : BOOLEAN);
(* Findet Element mit Schluessel 'SuchSchluessel':
   OK=TRUE : Element gefunden.
   OK=FALSE: Element war nicht in der Liste. *)

PROCEDURE RekSuchen (Baum: ZweigTyp);
BEGIN (* RekSuchen *)
  IF Baum = NIL THEN
    OK := FALSE;
  ELSE
    WITH Baum^ DO
      CASE StringVgl (SuchSchluessel, Inhalt.Schluessel) OF
        | Vor      : RekSuchen (UBLinks);
        | Nach     : RekSuchen (UBRechts);
        | Gleich   : Element := Inhalt;  OK := TRUE;
      END (* CASE *);
    END (* WITH *);
  END (* IF *);
END RekSuchen;

BEGIN (* Suchen *)
  RekSuchen (Gesamtbaum);
END Suchen;

(*****)
PROCEDURE Loeschen
  ( (* in *) LoeschSchluessel : SchluesselTyp;
    (* out *) VAR OK : BOOLEAN);

PROCEDURE RekLoeschen (VAR Baum : ZweigTyp);

PROCEDURE Entfernen ( VAR UB : ZweigTyp);
VAR Hilf: ZweigTyp;
BEGIN (* Entfernen *)
  IF UB^.UBRechts = NIL THEN
    Hilf := Baum;  Baum := UB;  UB := UB^.UBLinks;

```

```

    WITH Baum^ DO
        UBLinks := Hilf^.UBLinks;
        UBRechts := Hilf^.UBRechts;
    END (* WITH *);
    DEALLOCATE (Hilf, SIZE (KnotenTyp));
ELSE
    Entfernen (UB^.UBRechts);
END (* IF *);
END Entfernen;

VAR Hilf : ZweigTyp;
BEGIN (* RekLoeschen *)
    IF Baum = NIL THEN
        OK := FALSE;
    ELSE
        WITH Baum^ DO
            CASE StringVgl
                (LoeschSchluessel, Inhalt.Schluessel) OF
                | Vor      : RekLoeschen (UBLinks);
                | Nach     : RekLoeschen (UBRechts);
                | Gleich  : IF (UBLinks = NIL)
                            OR (UBRechts = NIL) THEN
                                Hilf := Baum;
                                IF UBRechts = NIL THEN
                                    Baum := UBLinks;
                                ELSE
                                    Baum := UBRechts;
                                END (* IF *);
                                DEALLOCATE (Hilf, SIZE (KnotenTyp));
                            ELSE
                                Entfernen (UBLinks);
                            END (* IF *);
                                OK := TRUE;
                        END (* CASE *);
                    END (* WITH *);
                END (* IF *);
            END RekLoeschen;

        BEGIN (* Loeschen *)
            RekLoeschen (Gesamtbaum);
        END Loeschen;

        (*****)

        BEGIN (* TabellenVerw *)
            Gesamtbaum := NIL;
        END BinSuchBaum.

```

Die durch Programm P6.8 verwaltete Tabelle könnte nun verwendet werden, um (bei entsprechend erweitertem Inhalt der Knotenelemente) ein Telefonbuch aufzunehmen. Das Verhalten eines solchen Programms wäre befriedigend, wenn der Baum "statistisch" wächst und schrumpft, d.h. annähernd *ausgeglichen* wie ein "natürlicher Baum" verzweigt ist.

Beispiel: 1.023 Knoten lassen sich im Idealfall völliger Ausgeglichenheit in einem Binärbaum der Höhe 10 anordnen, die maximale Pfadlänge von der Wurzel bis zu einem Blatt wäre dann also 9.

Baut man den Inhalt des Baums dagegen auf, indem man die Einträge aus einem existierenden Telefonbuch der Reihe nach übernimmt, so "entartet" der Baum zu einer *linearen Liste*.

Bei 1.023 Teilnehmern wird die Pfadlänge maximal 1.022.

Sofern wir das *Entarten* des Baums nicht verhindern (z.B. durch "*Balancieren*"), müssen wir wenigstens eine Rekursion verhindern, deren Tiefe mit der Zahl der Knoten im ungünstigsten Fall linear zunimmt; wir müssen also den rekursiven Algorithmus durch einen iterativen ersetzen.

Die durchaus zu bevorzugende Art rekursiver Programmierung können wir aber auf jeden Fall beibehalten, wenn wir durch geeignete Maßnahmen ausgeglichene Baumstrukturen mit einer gewissen *Höhenbeschränkung* garantieren. Im Idealfall (siehe obiges Beispiel) können wir in sogenannten vollständig ausgeglichenen Suchbäumen die Höhe *logarithmisch* bezüglich der Knotenzahl beschränken.

Ein Suchbaum heißt *vollständig ausgeglichen*: $\Leftrightarrow$

- für alle inneren Knoten des Baumes unterscheidet sich die Anzahl der Knoten in den Teilbäumen höchstens um 1
- alle Blätter des Baumes unterscheiden sich in ihrem Abstand zur Wurzel um höchstens 1.

Bei $n-1$ Knoten liegen die Blätter eines binären Suchbaums auf der Höhe $\lceil \lg n \rceil$.

Andere Definitionen vollständiger Ausgeglichenheit von Bäumen fordern, daß alle Blätter auf *gleicher* Höhe liegen. Dies ist aber nur durch Hinzufügen "virtueller" Knoten erreichbar, falls die Knotenzahl in einem binären Suchbaum ungleich $2^n - 1$ ist.

Der *Aufwand* zur Erhaltung vollständiger Ausgeglichenheit beim Einfügen und Löschen kann im ungünstigen Fall (wenn die einzufügenden oder zu löschenden Elemente wertemäßig dicht beieinander liegen) sehr hoch werden. In der Praxis entschärft man deshalb diese harte Forderung und wählt mit *schwächeren* Ausgeglichenheitsbegriffen und *lokalen*, d.h. auf Teilbäume eines Knotens bezogenen Höhenbeschränkungen einen *Kompromiß* zwischen

- *Reorganisationsaufwand* (beim Einfügen und Löschen!) und
- optimaler, logarithmischer *Suchzeit* bei vollständig ausgeglichenen Suchbäumen (mit allerdings i.a. hohen Balancieraufwand bei Einfüge- und Lösch-Operationen).

Solche relativ ausgeglichenen Bäume wollen wir nun in der Form sogenannter AVL-Bäume (AVL = Adelson-Velsky und Landis sind die Autoren der diese Bäume definierenden Veröffentlichung) kennenlernen.

6.4.1.2 AVL-Bäume

Ein (binärer) Suchbaum $T = (V, E)$ heißt *AVL-Baum* : $\Leftrightarrow$ für jeden Knoten $v \in V$ unterscheiden sich die Höhen der Unterbäume von v um höchstens 1.

Anschaulich: Knoten v habe m direkt anhängende Teilbäume $T_1, \dots, T_m$.

Dann gilt: $|\text{lev}(T_i) - \text{lev}(T_j)| \leq 1$ für alle $i, j \in \{1, 2, \dots, m\}$.

Im weiteren wollen wir uns auf *binäre AVL-Bäume* beschränken, ohne dies immer explizit zu sagen. Zur Unterstützung der einzelnen Operationen wird jedem Knoten eines AVL-Baumes eine Komponente neben Schlüssel und Inhalt hinzugefügt, der sogenannte *Balancefaktor*.

Sei $T = (V, E)$ ein AVL-Baum, $v \in V$.

$b_v \in \{-1, 0, 1\}$ heißt der *Balancefaktor* des Knotens v : $\Leftrightarrow$

$b_v = \text{Höhe des rechten Unterbaums von } v - \text{Höhe des linken Unterbaums von } v$.

Abb. 6.26 zeigt ein *Beispiel* eines AVL-Baumes, an dessen Knoten jeweils der entsprechende Balancefaktor notiert ist.

Abb. 6.26 AVL-Baum mit Balancefaktoren

Die Standard-Operationen des Einfügens und Löschens (wie wir sie in Programm P6.8 kennengelernt haben) können nun natürlich nicht mehr unverändert von den üblichen binären Suchbäumen übernommen werden, da der Betrag des Balancefaktors jedes Knotens nicht über 1 wachsen darf. Ein so modifiziertes, *balanciertes* Einfügen und Löschen für AVL-Bäume werden wir nun ausführlich behandeln.

Einfügen in AVL-Bäume

Wir gehen in fünf Schritten vor:

- zuerst ein motivierendes Beispiel,
- dann eine Generalisierung durch eine vollständige Fallunterscheidung,
- dann ein Algorithmusentwurf,
- dann eine Prozedur für das Einfügen (in enger Anlehnung an den Entwurf) und
- abschließend eine vollständige, elegantere MODULA-2 Implementierung für das Einfügen und alle Balancierungs-Operationen (die so auch später beim Löschen genutzt werden können).

Ein *Beispiel* (Einfügen von 5 in AVL-Baum aus Abb. 6.26) zeigt Abb. 6.27. Das Element 5 wird linker Nachfolger von 6 und reduziert die Balancefaktoren des Elements 6 auf -1 und des Elements 7 auf -2, womit die AVL-Eigenschaft verletzt ist.

Abb. 6.27 Durch Einfügen von 5 "verletzter" AVL-Baum

Durch eine sogenannte *Rechts-Rotation* des Unterbaums mit der Wurzel 7 entsteht wieder ein AVL-Baum (vgl. Abb. 6.28): die Wurzel "rotiert" (dreht ab) nach rechts und zieht den linken Sohn (und das daran hängende Blatt) hoch. Die Wurzel ändert ihren Balancefaktor von -2 auf 0, das Blatt bleibt unverändert und der "alte" Sohn (und "neue" Vater) erhöht den Balancefaktor von -1 auf 0.

Diesen Vorgang nennen wir *Rechts-Rotation*, weil die beteiligten Knoten (in diesem Fall die Knoten 5, 6, 7) alle eine *Rechtsdrehung* (also eine Drehung im Uhrzeigersinn) vollziehen. Analog werden wir später in diesem Abschnitt noch die *Links-Rotation* sehen. Eine alternative Bezeichnung (z.B. von Wirth gewählt) nennt die

Rechts-Rotation Links-Links-Rotation, da der Knoten, der einen unzulässigen Balancefaktor verursacht, nach zweimaligem Linksabstieg erreicht wird.

Abb. 6.28 Ergebnis der Rechts-Rotation um den Knoten 7

Unabhängig von dem vorgestellten Beispiel gilt für das Einfügen in einen AVL-Baum generell:

- wie bei den binären Suchbäumen fügen wir das neue Element als *Blatt* ein;
- der Balancefaktor kann nur längs des *Suchpfads* (von der Einfügestelle zur Wurzel) verändert worden sein;
- man suche den untersten Knoten v , an dem der Balancefaktor den Wert -2 oder $+2$ angenommen hat; falls es eine solche Stelle nicht gibt, hat der Baum nach wie vor die AVL-Eigenschaft; sonst gibt es einen der nachfolgend vorgestellten vier Fälle.

Falls es durch Einfügen eines Blattes x zur Verletzung der AVL-Eigenschaft gekommen ist, unterscheiden wir

1. Fall: Überhang im linken Teilbaum links von v
(x wird an α angehängt $\Rightarrow$ *R-Rotation*, siehe Abb. 6.29)
2. Fall: Überhang im "mittleren" Teilbaum links von v
(Reaktion: x wird an β (oder γ) angehängt $\Rightarrow$ *Links-Rechts-Rotation*, siehe Abb. 6.30)
3. und 4. Fall:
Die Fälle 3 und 4 sind symmetrisch zu 1 und 2, d.h. falls $b_v = +2$ ist, dann hängt entweder der rechte Teilbaum rechts von v über ($\Rightarrow$ *Links-Rotation*), oder im "mittleren" Teilbaum rechts von v hängt der linke oder rechte Teilbaum über ($\Rightarrow$ *Rechts-Links-Rotation*).

Abb. 6.29 Rechts-Rotation um Knoten v

Abb. 6.30 Links-Rechts-Rotation um Knoten u und dann w

Bemerkungen zu den Abbildungen:

- Im AVL-Baum vor der Rotation sind *zwei* Balancefaktoren angegeben. Links vom Knoten der Faktor *vor*, rechts vom Knoten der Faktor *nach* dem Einfügen des Elementes *x*.
- Die gestrichelten Linien sind Höhenlinien und dienen der Orientierung.

Für das Reorganisieren nach Verletzung der AVL-Eigenschaft dürfen wir für die vier Fälle zusammenfassend feststellen:

- Nach nur *einer* Rotation ist bereits der gesamte Baum wieder ausgeglichen.
- Die Knoten *u*, *v*, *w* werden jeweils *vertikal*, d.h. unter Wahrung ihrer Schlüsselordnung verschoben, wobei nur Zeiger umgesetzt werden. Wegen der Suchbaumeigenschaft gibt es keine *horizontale* Bewegung, die die Ordnung auf den Schlüssel zerstören würde.

Algorithmusentwurf (Einfügen in AVL-Baum):

- (1) Füge das neue Element *x* als Blatt ein.
- (2) Gehe zum Vaterknoten zurück (dieser sei *y*, *b_y* dessen Balancefaktor):
 - (a) Falls *x* im linken Unterbaum von *y* eingefügt wurde
 - (a1) Falls *b_y* = 1, dann *b_y* := 0, weiter mit (3);
 - (a2) Falls *b_y* = 0, dann *b_y* := -1, weiter mit (2);
 - (a3) Falls *b_y* = -1, dann entweder *R- oder LR-Rotation*; weiter mit (3).
 - (b) Falls *x* im rechten Unterbaum von *y* eingefügt wurde
 - (b1) Falls *b_y* = -1, dann *b_y* := 0, weiter mit (3);
 - (b2) Falls *b_y* = 0, dann *b_y* := 1, weiter mit (2);
 - (b3) Falls *b_y* = 1, dann entweder *L- oder RL-Rotation*; weiter mit (3).
- (3) Gehe zurück zur Wurzel; Ende.

Algorithmus:

Die folgende rekursive Prozedur *Insert* realisiert das Einfügen in einen AVL-Baum entsprechend dem oben dargestellten Algorithmusentwurf. Sie entspricht nicht Anforderungen des modernen Software Engineering, zeigt jedoch, daß die obige Fallunterscheidung direkt in einen Algorithmus umgesetzt werden kann. Die Wahl zwischen den Fällen (2) und (3) wird dabei durch den Resultat-Parameter *up* gesteuert. Eine übersichtlichere Implementierung des Einfügens in einem AVL-Baum folgt mit Programm P6.10 weiter unten in diesem Abschnitt.

```

PROCEDURE Insert      (      x      : INTEGER;
                        VAR ptr     : NodePtr;
                        VAR up      : BOOLEAN);

```

(* Programm P6.9 *)

(* Suchen , Einfügen und Ausgleichen im AVL - Baum *)

```

PROCEDURE RRotation (VAR ptr : NodePtr;
(* R-Rotation und entsprechende Korrektur des Balance-
faktors *)
VAR ptr1 : NodePtr;
BEGIN (* RRotation *)
    ptr1 := ptr^.left;
    ptr^.left := ptr1^.right;
    ptr1^.right := ptr;
    ptr^.bal := 0;
    ptr := ptr1;
END RRotation;

```

```

PROCEDURE LRRotation (VAR ptr : NodePtr;
(* LR-Rotation und entsprechende Korrektur der Balance-
faktoren *)
VAR ptr1 , ptr2 : NodePtr;
BEGIN (* LRRotation *)
    ptr1 := ptr^.left;
    ptr2 := ptr1^.right;
    ptr1^.right := ptr2^.left;
    ptr2^.left := ptr1;
    ptr^.left := ptr2^.right;
    ptr2^.right := ptr;
    IF ptr2^.bal = -1 THEN
        ptr^.bal := 1;
    ELSE
        ptr^.bal := 0;
    END (* IF *);
    IF ptr2^.bal = +1 THEN
        ptr1^.bal := -1;
    ELSE
        ptr1^.bal := 0;
    END (* IF *);
    ptr := ptr2;
END LRRotation;

```

```

PROCEDURE LRotation (VAR ptr : NodePtr;
(* L-Rotation und entsprechende Korrektur des Balance-
faktors *)
VAR ptr1 : NodePtr;
BEGIN (* LRotation *)
    ptr1 := ptr^.right;
    ptr^.right := ptr1^.left;
    ptr1^.left := ptr;
    ptr^.bal := 0;
    ptr := ptr1;
END LRotation;

```

```

PROCEDURE RLRotation (VAR ptr : NodePtr;
(* RL-Rotation und entsprechende Korrektur der Balance-
faktoren *)
VAR ptr1 , ptr2 : NodePtr;

```

```

BEGIN (* LRRotation *)
    ptr1 := ptr^.right;
    ptr2 := ptr1^.left;
    ptr1^.left := ptr2^.right;
    ptr2^.right := ptr1;
    ptr^.right := ptr2^.left;
    ptr2^.left := ptr;
    IF ptr2^.bal = +1 THEN
        ptr^.bal := -1;
    ELSE
        ptr^.bal := 0;
    END (* IF *);
    IF ptr2^.bal = -1 THEN
        ptr1^.bal := 1;
    ELSE
        ptr1^.bal := 0;
    END (* IF *);
    ptr := ptr2;
END RLRotation;

(* ptr^.bal = 1 *)
(* ptr^.right^.bal = -1 *)

BEGIN (* Insert *)
    IF ptr = NIL THEN
        ptr := ALLOCATE (ptr, SIZE(Node));
        up := TRUE;
        WITH ptr^ DO
            key := x;
            count := 1;
            left := NIL;
            right := NIL;
            bal := 0;
        END (* WITH *);
    ELSIF ptr^.key > x THEN
        Insert(x, ptr^.left, up);
    IF up THEN
        CASE ptr^.bal OF
            | 1: ptr^.bal := 0;
            | 0: ptr^.bal := -1;
            | -1: up := FALSE;
        END (* CASE *);
        IF ptr^.left^.bal = -1 THEN
            RRotation (ptr);
        ELSE
            LRRotation (ptr);
        END (* IF *);
        ptr^.bal := 0;
    END (* IF *);
    ELSIF ptr^.key < x THEN
        Insert(x, ptr^.right, up);
    IF up THEN
        CASE ptr^.bal OF
            | -1: ptr^.bal := 0;
            | 0: ptr^.bal := 1;
        END (* CASE *);
    END (* IF *);
END (* Insert *)

(* einfügen *)
(* Punkt (1) aus Entwurf *)
(* x muß in linken Teilbaum *)
(* Punkt (2a) aus Entwurf *)
(* Fall (a1) *)
(* Fall (a2) *)
(* Fall (a3) *)
(* Punkt (2a) abgeschlossen *)
(* x muß in rechten Teilbaum *)
(* Punkt (2b) aus Entwurf *)
(* Fall (b1) *)
(* Fall (b2) *)

```



```

      | 1:  up := FALSE;                      (* Fall (b3)      *)
          IF ptr^.right^.bal = 1 THEN
              LRotation (ptr); *)
          ELSE
              RLRotation (ptr);
          END (* IF *);
          ptr^.bal := 0;
      END (* CASE *);
  END (* IF *);                      (* Punkt (2b) abgeschlossen *)
ELSE                                (* x schon vorhanden *)
  INC(ptr^.count);
END (* IF *);
END Insert;

```

Gestartet wird jeweils mit Insert (element, wurzel, aufwärts), wobei aufwärts den Wert FALSE besitzt, element das einzufügende Element ist und wurzel auf die Wurzel des AVL-Baumes zeigt.

Modula-2-Implementierung eines AVL-Baumes

Nachfolgend ist die Modula-2-Realisierung der wichtigen Rotationsprozeduren sehr ausführlich wiedergegeben (eine effizientere Version ist z.B. im Buch von Wirth: "Algorithmen und Datenstrukturen in MODULA-2"; siehe Literaturverzeichnis). Hinzu kommen Prozeduren für das Ausgeben eines AVL-Baumes und für die Standard-Operation des Einfügens eines Elementes.

```

MODULE AVLBaum;                                (* Programm P6.10 *)
FROM InOut  IMPORT  ReadInt, Write, WriteInt, WriteString,
                    WriteLn;
FROM Storage IMPORT  ALLOCATE, DEALLOCATE;
TYPE ZweigTyp  = POINTER TO KnotenTyp;
   BalanceTyp = (LinksL, Neutral, RechtsL);
   KnotenTyp  = RECORD
       Schluessel : INTEGER;
       Balance    : BalanceTyp;
       UBLinks,  UBRechts : ZweigTyp;
   END (* RECORD *);
VAR Gesamtbaum : ZweigTyp;
  (*****)

PROCEDURE BaumAusgeben (Baum: ZweigTyp; Tiefe: INTEGER);

  PROCEDURE GibBalance;
  BEGIN (* GibBalance *)
    CASE Baum^. Balance OF
      | LinksL  : WriteString (' L');
      | RechtsL : WriteString (' R');
      | Neutral : WriteString (' =');
    END (* CASE *);
    WriteString ('          -- ');
  END GibBalance;

  VAR i: INTEGER;

```

```

BEGIN (* BaumAusgeben *)
  IF Baum # NIL THEN
    WITH Baum^ DO
      BaumAusgeben (UBRechts, Tiefe+1);
      FOR i := 1 TO Tiefe DO WriteString (' '); END;
      GibBalance; WriteInt (Schluessel, 1); WriteLn;
      BaumAusgeben (UBLinks, Tiefe+1);
    END (* WITH *);
  END (* IF *);
END BaumAusgeben;
(*-----*)

PROCEDURE Einfuegen ( (* inout *) VAR Baum : ZweigTyp;
                      (* in *) Wert : INTEGER;
                      (* inout *) VAR Hoeher : BOOLEAN);

  PROCEDURE Tausch (VAR Z1, Z2, Z3: ZweigTyp;
                    LR: ARRAY OF CHAR);
    (* Zyklisches Tauschen Z1 := Z2 usw.,
       Meldung dazu ausgeben *)
    VAR Hilf : ZweigTyp;
  BEGIN (* Tausch *)
    Hilf := Z1; Z1 := Z2; Z2 := Z3; Z3 := Hilf;
    WriteString (' ');
    WriteString (LR); WriteString ('rotation bei Knoten ');
    WriteInt (Hilf^.Schluessel, 1);
  END Tausch;

  PROCEDURE RotierenRechts (VAR Z: ZweigTyp);
  BEGIN (* RotierenRechts *)
    Tausch (Z, Z^.UBLinks, Z^.UBLinks^.UBRechts, 'Links');
  END RotierenRechts;

  PROCEDURE RotierenLinks (VAR Z: ZweigTyp);
  BEGIN (* RotierenLinks *)
    Tausch (Z, Z^.UBRechts, Z^.UBRechts^.UBLinks, 'Rechts');
  END RotierenLinks;

  PROCEDURE KorrBalanceUBe (Knoten: ZweigTyp);
    (* aufgrund der Symmetrie rechts wie links gueltig *)
    PROCEDURE SetBal (NeuLinks, NeuRechts: BalanceTyp);
    BEGIN (* SetBal *)
      WITH Knoten^ DO
        UBLinks^.Balance := NeuLinks;
        UBRechts^.Balance := NeuRechts;
      END (* WITH *);
    END SetBal;

  BEGIN (* KorrBalanceUBe *)
    CASE Knoten^.Balance OF
      | RechtsL : SetBal (LinksL, Neutral);
      | LinksL : SetBal (Neutral, RechtsL);
      | Neutral : SetBal (Neutral, Neutral);
    END (* CASE *);
  END KorrBalanceUBe;

```

```

PROCEDURE HoeherLinks;
BEGIN (* HoeherLinks *)
  CASE Baum^.Balance OF
    | LinksL : IF Baum^.UBLinks^.Balance = LinksL THEN
      RotierenRechts (Baum);      (* R-Rotation *)
      Baum^.UBRechts^.Balance := Neutral;
    ELSE
      (* LR-Rotation *)
      RotierenLinks (Baum^.UBLinks);
      RotierenRechts (Baum);      (* veraendert Baum *)
      KorrBalanceUBe (Baum);
    END (* IF *);
    Baum^.Balance := Neutral;    Hoeher := FALSE;
  | RechtsL : Baum^.Balance := Neutral;    Hoeher := FALSE;
  | Neutral : Baum^.Balance := LinksL;
  END (* CASE *);
END HoeherLinks;

PROCEDURE HoeherRechts;
BEGIN (* HoeherRechts *)
  CASE Baum^.Balance OF
    | RechtsL : IF Baum^.UBRechts^.Balance = RechtsL THEN
      RotierenLinks (Baum);      (* L-Rotation *)
      Baum^.UBLinks^.Balance := Neutral;
    ELSE
      (* RL-Rotation *)
      RotierenRechts (Baum^.UBRechts);
      RotierenLinks (Baum);      (* veraendert Baum *)
      KorrBalanceUBe (Baum);
    END (* IF *);
    Baum^.Balance := Neutral;    Hoeher := FALSE;
  | LinksL : Baum^.Balance := Neutral;    Hoeher := FALSE;
  | Neutral : Baum^.Balance := RechtsL;
  END (* CASE *);
END HoeherRechts;

BEGIN (* Einfuegen *)
  IF Baum = NIL THEN      (* neues Element in Baum einfuegen *)
    ALLOCATE (Baum, SIZE (KnotenTyp));
    WITH Baum^ DO
      UBLinks := NIL;    UBRechts := NIL;
      Schluessel := Wert;    Balance := Neutral;
    END (* WITH *);
    Hoeher := TRUE;      (* Mitteil. an hoehere Inkarnation
                          von Eintragen *)
  ELSE
    IF Wert = Baum^.Schluessel THEN
      WriteString ('    bereits im Baum vorhanden');
      WriteLn;
      Hoeher := FALSE;
    ELSIF Wert < Baum^.Schluessel THEN
      Einfuegen (Baum^.UBLinks, Wert, Hoeher);
      IF Hoeher THEN    HoeherLinks;    END;
    ELSE
      Einfuegen (Baum^.UBRechts, Wert, Hoeher);
  END

```



```

        IF Hoeher THEN    HoeherRechts;  END;
      END (* IF *);
    END (* IF *);
  END Einfuegen;
  (*****)

VAR Zahl : INTEGER;    Dummy : BOOLEAN;
BEGIN (* AVLBaum *)
  Gesamtbaum := NIL;
  WriteString ('Start AVLBaum');  WriteLn;  WriteLn;
  WriteString ('"L, =, R" bedeuten linkslastig, ');
  WriteString ('ausgeglichen, rechtslastig. ');
  WriteLn;  WriteLn;
  WriteString ('Gib nat. Zahlen fuer Baum, Abschluss mit 0');
  WriteLn;
  LOOP
    WriteString ('? ');  ReadInt (Zahl);
    IF Zahl = 0 THEN EXIT;  END;
    IF Zahl < 0 THEN
      WriteString (' bitte natuerliche Zahlen');  WriteLn;
    ELSE
      Einfuegen (Gesamtbaum, Zahl, Dummy);  WriteLn;
      WriteLn;  BaumAusgeben (Gesamtbaum, 0);  WriteLn;
    END (* IF *);
  END (* LOOP *);
  WriteLn;  WriteString ('Ende AVLBaum');  WriteLn;
END AVLBaum.

```

Löschen in AVL-Bäumen

Das Löschen in AVL-Bäumen geschieht zunächst wie bei binären Suchbäumen, d.h. das zu löschende Element wird durch seinen Inorder-Nachfolger ersetzt.

Nach der Lösch-Operation muß allerdings - beginnend ab der untersten veränderten Stelle - der Baum längs des Pfades zur Wurzel neu ausgeglichen werden, wobei u.U. - im Gegensatz zum Einfügen - *mehrere* Rotationen notwendig sind.

Wir betrachten hierzu einen Knoten u , dessen *linker* Teilbaum durch Löschen eines Elementes um 1 in der Höhe verringert wurde (analog für rechte Teilbäume), und können dabei durch Betrachten des Balancefaktor b_u drei Fälle (mit drei Spezialfällen für den 3. Fall) unterscheiden.

- Fall 1:* Falls $b_u = -1$,
dann $b_u := 0$ und mache mit dem *Vorgänger* von u weiter.
- Fall 2:* Falls $b_u = 0$,
dann $b_u = +1$ und *beende* das Ausgleichen.
- Fall 3:* Falls $b_u := +1$ und v der rechte Nachfolger von u ist,
dann unterscheide die *Spezialfälle*
3a: ($b_v = +1$), 3b: ($b_v = 0$), 3c: ($b_v = -1$).

Fall 3a: $b_v = +1$, deshalb *L-Rotation* von u (vgl. Abb. 6.31).

Dadurch wird die Gesamthöhe des Teilbaums (unter u bzw. v) um 1 erniedrigt. Daher muß man mit dem Vorgänger von v *weitermachen* und dessen Balancefaktor prüfen.

Abb. 6.31 Fall 3a beim Löschen im AVL-Baum: L-Rotation von u

Fall 3b: $b_v = 0$, deshalb *L-Rotation* von u (vgl. Abb. 6.32).

Die Höhe des Teilbaumes (unter u bzw. v) bleibt unverändert. Das Ausgleichen kann daher *beendet* werden.

Abb. 6.32 Fall 3b beim Löschen im AVL-Baum: L-Rotation von u

Fall 3c: $b_v = -1$ und w sei der linke Nachfolger von v mit $b_w = q \in \{-1, 0, +1\}$, je nach Wert für q fehlen in β oder γ eine Stufe ($q = -1$ oder $q = +1$) oder auch keine ($q = 0$). Dann *RL-Rotation* von v (Rechts-) und nachfolgend w (Links-). Abb. 6.33 zeigt diese Rotation: $a \in \{0, -1\}$ und $b \in \{0, +1\}$ geben die Balancefaktoren der Teilbäume mit den Wurzeln u bzw. v an. Da β und γ im gesamten Teilbaum mit der neuen Wurzel w um 1 Stufe gehoben wurden, verringert sich die Gesamthöhe um 1. Daher weitermachen beim Vorgänger von w .

Abb. 6.33 Fall 3c beim Löschen im AVL-Baum: RL-Rotation von v bzw. w .

Aufwandsabschätzungen

Einfügen und Löschen kosten im *schlechtesten* bzw. im *durchschnittlichen* Fall soviel Zeit, wie die *maximale* bzw. *mittlere* Höhe eines AVL-Baums mit n Knoten beträgt.

Empirische Resultate von *Aufwandsabschätzungen*:

- *Mittlere Höhe* eines AVL-Baumes mit n Knoten: ca. $\log(n) + 0.25$ (also fast so gut wie vollständig ausgeglichene Bäume).
- Wahrscheinlichkeit einer Rotation beim *Einfügen*: ca. 50%.
- Wahrscheinlichkeit einer Rotation beim *Löschen*: nur ca. 20%.

Für gegebene Knotenzahl n schätzen wir den *maximalen* Einfüge- und Löschaufwand (*worst case*) durch die größte mögliche Höhe eines n -knotigen AVL-Baumes. Wir können aber "umgekehrt" auch so vorgehen, daß wir für eine feste Baumhöhe r die *minimale* Knotenzahl bestimmen.

Wir interessieren uns also mit zwei Blickrichtungen für das Verhältnis von Knotenzahl zu Baumhöhe.

1. Blickrichtung: Gegebene Baumhöhe $\rightarrow$ minimale Knotenzahl

Sei $r \in \mathbb{N}_0$. Mit $\min_r$ bezeichnen wir die minimale, mit $\max_r$ die maximale Knotenzahl eines AVL-Baums der Höhe r . Es gilt: $\max_r = 2^{r-1}$. Wir interessieren uns nachfolgend aber nur für $\min_r$ und die daraus resultierenden AVL-Bäume.

Für $\min_r$ gilt: $\min_0 = 0$ und $\min_1 = 1$.

Im allgemeinen Fall haben wir bezüglich einer Wurzel w mit $b_w = 1$ im linken Teilbaum mindestens $\min_{r-2}$ Knoten, im rechten Teilbaum mindestens $\min_{r-1}$ Knoten (bzw. umgekehrt).

Dann gilt für den gesamten Baum:

$$\min_r = \min_{r-1} + \min_{r-2} + 1.$$

Betrachten wir die *Fibonacci-Zahlen* F_i (entsprechende Baumstrukturen, die sogenannten Fibonacci-Bäume führen - wie wir sehen werden - bei gegebener Baumhöhe zu minimaler Knotenzahl):

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, \dots, F_n = F_{n-1} + F_{n-2}.$$

Fibonacci-Bäume der Höhe i :

- Der leere Baum ist der Fibonacci-Baum der Höhe 0: FB_0 .
- Der ein-knotige Baum ist der Fibonacci-Baum der Höhe 1: FB_1 .
- Sind FB_{i-1} und FB_{i-2} die Fibonacci-Bäume der Höhe $i-1$ und $i-2$, dann ist der Baum (x, FB_{i-1}, FB_{i-2}) , d.h. x ist Vater der Teilbäume FB_{i-1} und FB_{i-2} , der Fibonacci-Baum der Höhe i : FB_i .

Die schrittweise Konstruktion von Fibonacci-Bäumen (der Höhe i) zeigt Abb. 6.34, wobei kursiv gekennzeichnet immer die Vorgänger-Bäume der Höhen $i-1$ und $i-2$ eingehen.

Abb. 6.34 Konstruktion der FB_i (Fibonacci-Bäume der Höhe i) bis $i = 5$

Die Anzahl der Knoten eines Fibonacci-Baumes der Höhe i entspricht also genau der um 1 erhöhten i -ten Fibonacci-Zahl (bei der Baumkonstruktion kommt immer noch eine Wurzel, also ein Knoten hinzu). Damit können wir nun den Zusammenhang zwischen der minimalen Knotenzahl eines AVL-Baumes und den Fibonacci-Bäumen herstellen.

Behauptung: $\min_r = F_{r+2} - 1$.

Beweis (durch vollständige Induktion über r):

Die Fälle $r = 0$ und $r = 1$ sind klar: $\min_0 = 0$, $\min_1 = 1$.

Für $r > 1$:

$$\begin{aligned}
 \min_r &= \min_{r-1} + \min_{r-2} + 1 \\
 &= (F_{r+1} - 1) + (F_r - 1) + 1 && \text{(Induktionsannahme)} \\
 &= F_{r+2} - 1 && \text{(nach Definition der Fibonacci-Zahlen).}
 \end{aligned}$$

Zu gegebener Baumhöhe sind also entsprechende Fibonacci-Bäume die AVL-Bäume mit jeweils minimaler Knotenzahl, d.h. die "entartetsten" oder auch "schiefe" AVL-Bäume.

2. Blickrichtung: Gegebene Knotenzahl $\rightarrow$ Baumhöhen

Es sei T_n ein AVL-Baum mit n Knoten. Zwischen welchen Grenzen liegt seine Höhe?

Hilfssatz: $F_n = \frac{1}{\sqrt{5}} (\varphi^n - \varphi^{-n})$ mit $\varphi = \frac{1+\sqrt{5}}{2}$ und $\varphi^{-1} = \frac{1-\sqrt{5}}{2}$

Beweis (durch vollständige Induktion über n):

Die Fälle $n = 0$ und $n = 1$ sind klar. Für $n > 1$ gilt:

$$\begin{aligned} F_{n+1} &= F_n + F_{n-1} \\ &= \frac{1}{\sqrt{5}} (\varphi^n - \varphi^{-n}) + \frac{1}{\sqrt{5}} (\varphi^{n-1} - \varphi^{-(n-1)}) \\ &= \frac{1}{\sqrt{5}} (\varphi^{n-1} (\varphi+1) - \varphi^{-(n-1)} (\varphi^{-1}+1)), \text{ wegen } \varphi+1=\varphi^2 \text{ und } \varphi^{-1}+1=\varphi^{-2} \text{ folgt daraus} \\ &= \frac{1}{\sqrt{5}} (\varphi^{n+1} - \varphi^{-(n+1)}). \end{aligned}$$

Wegen $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618034$ und $\varphi^{-1} = \frac{1-\sqrt{5}}{2} \approx -0.618$ geht der Subtrahend für höhere n gegen 0 und es gilt:

$$F_n = \text{round} \left(\frac{\varphi^n}{\sqrt{5}} \right); \text{ speziell } \min_r = \text{round} \left(\frac{\varphi^{r+2}}{\sqrt{5}} \right) - 1,$$

wobei round zur nächstliegenden ganzen Zahl rundet.

Wir haben bereits für $r = h(T_n) \geq 1$ gezeigt:

$$\frac{\varphi^{r+2}}{\sqrt{5}} - 2 < F_{r+2} - 1 = \min_r \leq n \leq \max_r = 2^r - 1.$$

Daraus folgt: $(r+2) \cdot \log(\varphi) - \log(\sqrt{5}) < \log(n+2) < r+1$

Durch Umformen erhalten wir für die

- rechte Ungleichung: $r > \log(n+2) - 1.$
- linke Ungleichung: $r < \frac{1}{\log \varphi} \cdot \log(n+2) + \frac{\log \sqrt{5}}{\log \varphi} - 2$

Durch Ausrechnen der konstanten Faktoren folgt dann:

$$r < 1.4404 \log(n+2) - 0.328.$$

Es sei r die Höhe eines AVL-Baumes mit n Knoten. Dann gilt:
 $\log(n+2) - 1 < r < 1.4404 \log(n+2) - 0.328.$

Die *maximale Suchzeit* liegt bei einem AVL-Baum also höchstens um gut 44% höher als bei einem vollständig ausgeglichenen Baum (die mittlere Suchzeit ist aber wesentlich günstiger). Sie ist also nie schlechter als $O(\log n)$.

6.4.1.3 Optimale Suchbäume

Bisher sind wir bei den vorgestellten Suchstrukturen (Listen, Suchbäume, AVL-Bäume) immer davon ausgegangen, daß die *Wahrscheinlichkeit*, mit der nach einem Schlüssel gesucht wird, unbeachtet bleiben kann (d.h. wir haben sie für alle Schlüssel als *gleich* unterstellt). Wenn man diese i.a. praxisfremde Vorstellung aufgibt, gewinnt die Platzierung eines Knotens in einer Baumstruktur erhebliche Bedeutung, da dessen Höhe mit den Kosten für seinen Zugriff korrespondiert (je weiter die Distanz zur Wurzel, desto teurer der Zugriff). Es liegt nahe, die häufig gesuchten Schlüssel nahe an der Wurzel unterzubringen, natürlich ohne die wichtige Suchbaumeigenschaft zu verletzen. Eine sinnvolle Anwendung optimaler Suchbäume ist z.B. die Schlüsselworttabelle eines *Compilers* oder Interpreters (Tabelle der Terminalsymbole), da die Terminalsymbole einer Programmiersprache sich nicht ändern (also eine konstante Datenmenge bilden) und unterschiedliche Zugriffswahrscheinlichkeiten besitzen, die empirisch ermittelt werden können.

Die Qualität eines so angelegten Baumes läßt sich messen, indem man die *Kosten* für jeden Knoten als Produkt seiner Distanz zur Wurzel und seiner Zugriffswahrscheinlichkeit berechnet und die *Gesamtkosten* des Baumes als Summe über die einzelnen Knotenkosten festlegt. Ein bezüglich dieser Kostenfunktion kostengünstigster Suchbaum (oder - falls nicht eindeutig - die Menge der *kostengünstigsten* Suchbäume) wird als *optimal* bezeichnet.

Wir werden die optimalen Suchbäume in einer Art einführen, die in den Knoten nicht nur die Schlüssel, sondern (in den Blättern!) auch die Intervalle für einerfolgloses Suchen repräsentiert.

Beispiel :

In den inneren Knoten eines binären Suchbaumes stehen die Schlüssel $K_1, K_2, K_3, \dots, K_n$ (als Repräsentanten für eine *erfolgreiche Suche*), während die Blätter vorhandene Schlüssel x (als Intervalle) darstellen (als Repräsentanten für eine *erfolglose Suche*).

Abb. 6.35 Beispiel eines binären Suchbaumes mit Blättern für erfolglose Suche

Es seien

$p_i =$ Wahrscheinlichkeit, daß nach K_i ($i = 1, \dots, n$) gesucht wird und
 $q_j =$ Wahrscheinlichkeit, daß das gesuchte Element x zwischen K_j und K_{j+1} ($j = 0, 1, \dots, n$) liegt, wobei q_0 dem Bereich " $< K_1$ " und q_n dem Bereich " $> K_n$ " zugeordnet sind,

mit der Eigenschaft ("Normierung" der Wahrscheinlichkeiten)

$$\sum_{i=1}^n p_i + \sum_{j=0}^n q_j = 1.$$

Sei $\text{lev}(i)$ Level des inneren Knotens i (mit Inhalt K_i) und $\text{lev}(j)$ Level des Blattes " $K_j < x < K_{j+1}$ ". Für einen gegebenen binären Suchbaum B ist dann die (mittlere) Suchzeit:

$$S_B = \sum_{i=1}^n p_i \text{lev}(i) + \sum_{j=0}^n q_j \text{lev}(j).$$

Aufgabe: Suche (zu gegebenen Schlüsseln $K_1, K_2, \dots, K_n$ und Wahrscheinlichkeiten für erfolgreiche und erfolglose Suche nach diesen Schlüsseln) einen binären Suchbaum B_0 mit den Eigenschaften

(a) B_0 ist binärer Suchbaum mit den Schlüsseln $K_1, K_2, \dots, K_n$ und

- (b) $S_{B_0} = \min \{S_B \mid B \text{ ist binärer Suchbaum mit den Schlüsseln } K_1, K_2, \dots, K_n\}.$

Jeder B_0 , der diese Eigenschaft besitzt, heißt *optimaler Suchbaum*, d.h. ein binärer Suchbaum mit minimaler mittlerer Suchzeit ist ein optimaler Suchbaum.

Beispiel:

Gegeben seien die Schlüssel 0, 5, 7, 11, die Zusicherung, daß nur mit ganzzahligen Argumenten gesucht wird, und die Wahrscheinlichkeiten:

$p_1 = \frac{1}{12}$ (für 0), $p_2 = \frac{1}{6}$ (für 5), $p_3 = \frac{1}{8}$ (für 7), $p_4 = \frac{1}{6}$ (für 11) für die erfolgreiche Suche sowie

$q_0 = \frac{1}{12}$ (für < 0), $q_1 = \frac{1}{8}$ (für 1..4), $q_2 = \frac{1}{24}$ (für 6), $q_3 = \frac{1}{12}$ (für 8..10) und $q_4 = \frac{1}{8}$ (für > 11) für die erfolglose Suche.

Eine 1. Variante (B_1) zur Konstruktion eines Suchbaums zeigt Abb. 6.36.

Abb. 6.36 Variante B_1 eines Suchbaums

Die Suchzeit für den Suchbaum B_1 beträgt

$$S_{B_1} = \frac{3}{12} + \frac{2}{12} + \frac{3}{8} + \frac{1}{6} + \frac{3}{24} + \frac{2}{8} + \frac{4}{12} + \frac{3}{6} + \frac{4}{8} = \frac{4}{6} + \frac{9}{8} + \frac{9}{12} + \frac{3}{24} = \frac{64}{24}.$$

Eine 2. Variante (B_2) sehen wir in Abb. 6.37.

Für B_2 errechnet sich die Suchzeit wie folgt:

$$S_{B_2} = \frac{3}{12} + \frac{2}{12} + \frac{4}{8} + \frac{3}{6} + \frac{4}{24} + \frac{1}{8} + \frac{3}{12} + \frac{2}{6} + \frac{3}{8} = \frac{5}{6} + \frac{8}{8} + \frac{8}{12} + \frac{4}{24} = \frac{64}{24}.$$

Beide Varianten haben zufällig die gleiche mittlere Suchzeit. Sie sind aber beide nicht optimal!

Abb. 6.37 Variante B₂ eines Suchbaums

Konstruktion optimaler Suchbäume

Die naheliegende Frage ist, wie man (möglichst effizient) zu gegebenen Schlüsseln und deren Zugriffswahrscheinlichkeiten einen optimalen Suchbaum findet. Ein *naiver* Ansatz besteht darin, alle zu gegebenen Schlüsseln $K_1, K_2, \dots, K_n$ gehörenden binären Suchbäume zu ermitteln und den mit der geringsten mittleren Suchzeit als den optimalen auszuwählen.

Aufwandsabschätzung: Es gibt ca. $\binom{2n}{n} \cdot \frac{1}{n+1}$,

d.h. ca. 4^n zu untersuchende Bäume. Also scheidet dieser Ansatz für praktisch interessante Probleme mit großem n aus.

Optimale Suchbäume haben aber eine bemerkenswerte Eigenschaft, die bei ihrer Konstruktion nützlich ist:

Wenn B_0 optimaler Suchbaum mit Wurzel w und Teilbäumen B_1 und B_2 ist, dann müssen auch B_1 und B_2 optimale Suchbäume (bzgl. der um w reduzierten Schlüsselmenge) sein!

Diese Eigenschaft empfiehlt einen Algorithmus, der eine *inkrementelle Konstruktion* eines optimalen Suchbaums realisiert, ausgehend von einzelnen Knoten als kleinstmögliche Teilbäume. Wir benötigen deshalb eine Notation für die Suchzeit in einem Teilbaum mit den Schlüsseln $K_{k+1}, K_{k+2}, \dots, K_m$.

Sei $B^{k,m}$ ($0 \leq k \leq m \leq n$) ein Teilbaum mit den Schlüsseln $K_{k+1}, K_{k+2}, \dots, K_m$ und $B_0^{k,m}$ ein entsprechender optimaler Suchbaum. Dann ist

$$c(k, m) = \sum_{i=k+1}^m p_i \text{lev}(i) + \sum_{j=k}^m q_j \text{lev}(j)$$

die mittlere Suchzeit des optimalen Suchbaums $B_o^{k,m}$.

c steht dabei als Abkürzung für das englische Wort "cost".

Das Verständnis für die Berechnung der $c(k,m)$ steckt in der Gleichung

$$S_{B_o^{k,m}} = p(k, m) + S_{B_L} + S_{B_R},$$

die besagt, daß die Suchzeit des Teilbaums $B_o^{k,m}$ gleich der Summe aus der Wahrscheinlichkeit für einen Zugriff auf $B_o^{k,m}$ und den Suchzeiten des rechten und linken Teilbaums von $B_o^{k,m}$ ist. Dabei gilt:

$$p(k, m) = \sum_{i=k+1}^m p_i + \sum_{j=k}^m q_j.$$

Abb. 6.38 Aufteilung des Suchbaums $B_o^{k,m}$ bzgl. Wurzel K_p

Während $p(k, m)$ für gegebene Parameter k und m bestimmt ist, hängen die Suchzeiten S_{B_L} und S_{B_R} von der Wahl der Wurzel des Teilbaums $B_o^{k,m}$ ab. Mit einer Wurzel K_p ($k < p \leq m$) hat $B_o^{k,m}$ dann die in Abb. 6.38 dargestellte Form.

Für die Bestimmung von $B_o^{k,m}$ muß also die Wurzel K_p bestimmt werden, für die die Summe der beiden mittleren Suchzeiten $c(k, p-1)$ und $c(p, m)$ innerhalb der Teilbäume $B_o^{k,p-1}$ bzw. $B_o^{p,m}$ minimal ist.

Dann gilt also:

$$c(k, m) = \sum_{i=k+1}^m p_i + \sum_{j=k}^m q_j + \min \{c(k, p-1) + c(p, m) \mid k < p \leq m\}$$

$$c(k, k) = q_k.$$

Schließlich gilt für die mittlere Suchzeit des optimalen Suchbaums $S_{B_0} = c(0, n)$.

Analyse

Es gibt genau $1/2 (n+1) (n+2)$ Werte für $c(k, m)$. Ihre Berechnung erfordert $O(n^2)$ *Speicherplatz* und folgenden *Zeitaufwand* ("m-k" ist die Anzahl möglicher p, die Addition "+1" ist für $c(k, k)$):

$$\begin{aligned} \sum_{k=0}^n \sum_{m=k}^n m-k+1 &= \sum_{k=0}^n \sum_{r=1}^{n-k+1} r = \sum_{k=0}^n \frac{1}{2} \cdot (n-k+1) (n-k+2) \\ &= \sum_{t=1}^{n+1} \frac{1}{2} \cdot t \cdot (t+1) = \frac{1}{2} \sum_{t=1}^{n+1} t^2 + \frac{1}{2} \cdot \frac{1}{2} \cdot (n+1) (n+2) \\ &= \frac{1}{2} \cdot \frac{1}{6} \cdot (n+1) (n+2) (2n+3) + \frac{1}{4} \cdot (n+1) (n+2) \\ &= \frac{1}{6} \cdot n^3 + O(n^2). \end{aligned}$$

Das vorgestellte Verfahren zur Berechnung optimaler Suchbäume mit Hilfe aller $c(k, m)$ -Werte benötigt also $O(n^3)$ Zeit, sofern die obige Rekursion verwendet wird. Allerdings kann die Laufzeit weiter verringert werden. Auf jeden Fall ist gegenüber dem naiven Ansatz mit exponentieller Laufzeit ein entscheidend günstigerer, nur noch *polynomialer* Algorithmus gefunden.

6.4.1.4 B-Bäume

Eine häufige Praxisanforderung an Suchbäume können die bisher vorgestellten Baumstrukturen nicht erfüllen: die effiziente Erstellung und die Reorganisation (Einfügen, Löschen, Sichern) von *sehr großen* Datenmengen, die nicht mehr im *Hauptspeicher* eines Rechners verwaltet werden können.

Wir betrachten also die Speicherung von Massendaten auf Hintergrundspeichern, z.B. auf einer Platte. Dazu verwendet man häufig sogenannte *B-Bäume* (oder auch

Vielweg-Bäume), die in den Knoten *mehrere* Schlüssel speichern und damit die Effizienz von Zugriffen auf externe Speichermedien wie Platten erhöhen. B-Bäume werden aber auch dann eingesetzt wenn, wenn nur die Blattebene (oder die beiden letzten Ebenen) auf der Platte sind, wobei die Hauptspeicherknoten nur Schlüssel, also keine Informationen enthalten. Das "B" steht dabei für "Balanced", also (höhen-)balanciert.

Bei der Auswahl eines Kriteriums für das "kontrollierte Wachsen" (um "Entartungen" mit zu tiefen Teilbäumen zu verhindern) eines B-Baums scheiden Kriterien wie vollständige Ausgeglichenheit oder auch nur AVL-Eigenschaft wegen ihres zu großen Verwaltungsaufwands aus. Man schwächt die Anforderungen auf ein nicht so anspruchsvolles Maß ab: grob gesagt verlangt man, daß jeder Knoten im Baum zwischen m und $2m$ Schlüssel für ein konstantes m hat.

Ein Baum heißt *B-Baum* (der Ordnung m) : $\Leftrightarrow$

- (a) Jeder Knoten außer der Wurzel enthält mindestens m Schlüssel.
- (b) Jeder Knoten enthält höchstens $2m$ Schlüssel.
- (c) Ein Knoten mit k Schlüsseln, der nicht Blatt ist, besitzt genau $k+1$ Söhne.
- (d) Alle Blätter besitzen den gleichen Level, d.h. stehen auf gleicher Stufe.
- (e) Sind $S_1, S_2, \dots, S_k$ mit $m \leq k \leq 2m$ die Schlüssel eines Knotens x , dann sind alle Schlüssel des 1. (d.h. kleinsten) Sohnes von x kleiner als S_1 , alle Schlüssel des $(k+1)$ -ten Sohnes größer als S_k und alle Schlüssel des i -ten Sohnes mit $1 < i < k+1$ größer als S_{i-1} und kleiner als S_i . Als *kleinsten Sohn* bezeichnen wir im folgenden den am weitesten links stehenden, direkten Sohnknoten.

Bemerkungen

1. B-Bäume (der Ordnung m) mit n Knoten haben höchstens die *Höhe* $\log_{m+1}(n)$.
2. Die *Speicherausnutzung* bei effizienter Implementierung ist $> 50\%$ (wegen (a) und (b)).
3. Die allgemeine Knotenstruktur eines B-Baumes der Ordnung m zeigt Abb. 6.39.

Erläuterungen:

$S_1 < S_2 < \dots < S_k$ sind *Schlüssel*, $p_0, \dots, p_k$ sind *Knotenverweise* mit $m \leq k \leq 2 \cdot m$

p_0 verweist auf Knoten mit Schlüsseln $s < S_1$

p_1 verweist auf Knoten mit Schlüsseln $s \in] S_1, S_2 [$ (offenes Intervall mit den Grenzen S_1 und S_2).

...

p_{k-1} verweist auf Knoten mit Schlüsseln $s \in] S_{k-1}, S_k [$

p_k verweist auf Knoten mit Schlüsseln $s > S_k$.

Abb. 6.39 Struktur eines Knotens in einem B-Baum

Wenn ein Datenbestand mit n Schlüsseln verwaltet werden soll, wird die Ordnung m des B-Baumes so gewählt, daß

$\log_{m+1}(n) \cdot$	“für maximale Baumhöhe addiere”
(Latenzzeit +	“Wartezeit für Positionierung auf den Bereich der Platte, auf den p_i zeigt”
Übertragungszeit +	“für maximal $4m+1$ Daten (= 1 Knoten)”
interne Verarbeitungszeit)	“das interne Suchen erfolgt z.B. binär, also mit $O(\log(m))$ Schritten”

minimal wird. Es ist klar, daß die konkreten Zeiten sehr stark von den technischen Charakteristika der Platten-Hardware und vom Betriebssystem abhängen (häufige Ordnung in der Praxis: $128 \leq m \leq 512$).

Suchen und Einfügen in B-Bäumen

Beim Suchen nach einem Schlüssel S in einem B-Baum holt man einen Knoten (entspricht i.a. einer Speicherseite oder auch kurz *Seite*) in den Hauptspeicher und prüft, ob der gesuchte Schlüssel K im Knoten ist. Wenn das der Fall ist, ist man fertig (d.h. mit dem Schlüssel S findet man zugleich den Verweis, wo das zugehörige Datenelement auf der Platte steht).

Falls man S nicht findet, so gibt es ein i , mit $S_i < S < S_{i+1}$ bzw. $S < S_1$ (dann ist $i = 0$) oder $S > S_k$ (dann ist $i = k$). Nun holt man den Knoten bzw. die Seite in den Hauptspeicher, auf den p_i zeigt, und setzt die Suche dort fort.

Hat man S auch in der Blattebene nicht gefunden, so fügt man es in den entsprechenden Knoten ein. Falls dabei ein “Überlauf” durch Einfügen des $(2m+1)$ -ten Elements droht, wird das mittlere, d.h. m -te Element in den Vaterknoten hochgezogen und der dadurch wieder auf $2m$ Schlüssel reduzierte Knoten “gespalten”, um die B-Baum-Eigenschaft zu bewahren (da der Vater nun einen Schlüssel mehr besitzt, muß ein neuer Sohn entstehen!). Die Höhe eines B-Baums ändert sich dabei nur, falls durch wiederholten Knotenüberlauf schließlich der Wurzelknoten gespalten werden muß (also eine neue Wurzel erzeugt wird), so daß Forderung (d) stets erfüllt ist.

Beispiel

Es sei eine Liste von INTEGER-Werten gegeben.
Folgender B-Baum ($m = 2$) sei bereits erstellt:

Abb. 6.40 B-Baum mit INTEGER-Werten

Folgende Werte sollen sequentiell unter Wahrung der B-Baum-Eigenschaft eingefügt werden: 16, 24, 23, 21, 20.

1. "16" einfügen

Abb. 6.41 B-Baum nach Einfügen von "16"

2. "24" einfügen =>
"Überlauf", deshalb mittleres Element hochziehen und Knoten "spalten".

Abb. 6.42 B-Baum nach Einfügen von "24" und Überlaufbehandlung

3. "23" einfügen

Abb. 6.43 B-Baum nach Einfügen von "23"

4. "21" einfügen

Abb. 6.44 B-Baum nach Einfügen von "21"

5. "20" einfügen =>

"doppelter Überlauf", d.h. zweimal mittleres Element hochziehen und splitten.

Abb. 6.45 B-Baum nach Einfügen von "20" und doppelter Überlaufbehandlung

Nach dieser grafischen Veranschaulichung soll nun der Algorithmus zum Suchen und Einfügen in B-Bäumen vorgestellt werden. Dabei können viele komplexe Abschnitte nur in Pseudocode (jeweils in Anführungszeichen) präsentiert werden.

Algorithmusentwurf:

```

(* Programm P6.11 *)
TYPE  SeitePtr = POINTER TO Seite;
      Item      = RECORD          (* Die Struktur eines Knotens *)
        schluessel : SchlüsselTyp;

```

```

        inhalt      : InhaltTyp;
        zeiger      : SeitePtr;
    END (* RECORD *);
Seite = RECORD (* Die Struktur einer Seite *)
    anzahl      : [0..2*m];
                (* Anzahl Schluessel im Knoten *)
    linkeSeite : SeitePtr;
                (* Verweis auf kleinste Seite *)
    items       : ARRAY [1..2*m] OF Item;
END (* RECORD *);

PROCEDURE BBaumsuche (    Schluessel : SchluesselTyp;
                        Inhalt      : InhaltTyp;
                        Seite       : SeitePtr;
                        VAR weiter  : BOOLEAN;
                        VAR Eintrag  : Item);
(* Schluessel ist der zu suchende Schluessel und Inhalt der
   dazugehoerige Inhalt, Seite die aktuell zu durchsuchende Seite
   und weiter ist genau dann TRUE, wenn das Item Eintrag nach oben
   weitergegeben werden muß *)

BEGIN (* BBaumsuche *)
    IF Seite = NIL THEN (* Schluessel ist nicht im B-Baum *)
        "Bilde Eintrag aus Schluessel, Inhalt und NIL.
        Setze weiter := TRUE, damit Eintrag eingefuegt wird"
    ELSE
        WITH Seite^DO
            "Suche Schluessel in der durch den Zeiger 'Seite' referen-
            zierten Seite mit binaerer Suche durch Vergleiche der
            Schluesselkomponenten von Items";
            IF "Schluessel gefunden" THEN (* kein Einfuegen noetig *)
                "verarbeite"
            ELSE
                BBaumsuche (Schluessel, "Folgezeiger", weiter, Eintrag);
                (* Evtl. (abh. von weiter) nach Rueckkehr Schluessel
                einfuegen *);
                IF weiter THEN
                    IF (anzahl < 2*m) THEN (* Seite hat noch Platz *)
                        "Fuege Eintrag auf Seite ein";
                        weiter := FALSE;
                    ELSE
                        (* Seite ist bereits voll *)
                        "Fuege Eintrag ein und gib mittleres Element im Para-
                        meter Eintrag zurueck.
                        Falls Seite nicht die Wurzelseite ist, zerlege die
                        Seite in zwei Seiten und entferne dabei das mittlere
                        Element."
                    END (* IF *);
                END (* IF *);
            END (* IF *);
        END (* WITH *);
    END (* IF *);
END BBaumsuche;

```

Der *erste Aufruf* von BBaumsuche erfolgt in einem Rahmenprogramm mit der Argumentliste (K_i, Wurzelverweis, weiter, Eintrag). Falls der Ergebnis-Parameter "weiter"

anschließend den Wert TRUE hat, so ist entweder der B-Baum vorher leer gewesen oder eine Zerlegung der Wurzelseite erforderlich; beide Fälle müssen durch das aufrufende Rahmenprogramm abgefangen werden. Der Ergebnis-Parameter Eintrag enthält dann das Item für die neu zu erzeugende Wurzelseite.

Bemerkung (zur Verbesserung des Algorithmus):

1. Statt bei einem *Überlauf* den überlaufenden Knoten w sofort in zwei Knoten w_1, w_2 zu splitten, könnte man auch die Inhalte eines *Bruderknotens* v einbeziehen und evtl. das Item diesem Bruderknoten zuschlagen. Nur wenn auch v bereits $2 \cdot m$ Elemente besitzt, wird so aufgespalten, daß v und w_1 und w_2 jeweils $4m/3$ Schlüssel enthalten.

Folglich ist dann mindestens $(4m/3)/2m$ des *Speicherplatzes* eines Knotens stets gefüllt, d.h. mindestens $2/3$ (statt $1/2$ wie bei B-Bäumen). Bäume mit dieser Eigenschaft heißen *B*-Bäume*.

Achtung: Es gibt auch Definitionen von B^* -Bäumen (z.B. im Bereich der Datenbanksysteme), die Informationen zu den Schlüssel nur in den Blättern zulassen. Durch eine zusätzliche Verkettung der Blätter ist in diesen B-Bäumen dann außerdem ein sequentieller Durchlauf durch die Schlüssel- und Informationsmenge möglich.

2. *Zugriffsanalyse:* Tiefe des B-Baumes $\leq \log_{m+1}(n)$.
Wie oft geschieht aber beim Einfügen ein *Aufspalten*?

Sei k die Anzahl der Knoten und n die Anzahl der Einträge in einem B-Baum. Dann gilt aufgrund der B-Baum Eigenschaft die Ungleichung:

$$m \cdot (k-1) + 1 \leq n \leq 2 \cdot m \cdot k.$$

Um aus einem Wurzelknoten einen B-Baum mit k Knoten zu erhalten, sind $k-1$ Aufspaltungen notwendig. Aus der obigen Ungleichung folgt deshalb für die Anzahl der Aufspaltungen:

$$\frac{n}{2m} - 1 \leq k - 1 \leq \frac{n-1}{m}.$$

Für die mittlere Anzahl der Aufspaltungen beim Einfügen folgt dann:

$$\frac{k-1}{n} \leq \frac{n-1}{n} \cdot \frac{1}{m} \leq \frac{1}{m},$$

also seltener als jedes m -te Mal eine Aufspaltung.

Löschen in B-Bäumen

Beim Löschen in B-Bäumen geht man so vor, daß man den Inorder-Nachfolger des zu löschenden Schlüssels S an die Stelle bringt, wo S stand, und am alten Ort löscht.

Algorithmusentwurf:

```

(* Programm P6.12 *)
PROCEDURE BBAumloesche (Schluessel : SchluesselTyp;
                       Seite: SeitePtr);
VAR q, Nachfolger : SeitePtr;
    NachfolgerSchluessel : SchluesselTyp;
    i : [0..2*m];
BEGIN (* BBAumloesche *)
    "Sei Schluessel = Seite^.items[i].schluessel";
    (* Schluessel ist i-ter Schluessel in seite *)
    q := Seite^.items[i].zeiger;
    Nachfolger := Seite;
    (* kleinster Schluessel unter q; suche Inorder-Nachfolger *)
    WHILE q # NIL DO
        Nachfolger := q;
        q := q.linkeSeite;
    END (* WHILE *);
    IF Seite = Nachfolger THEN (* d.h. Seite ist Blatt *)
        Loesche (Schluessel, Seite); (* Funktionalitaet s.u. *)
    ELSE (* Nachfolger verweist auf Inorder-Nachfolger *)
        NachfolgerSchluessel := Nachfolger^.items[1].schluessel;
        Seite^.items[i] := Nachfolger^.items[1];
        Loesche (NachfolgerSchluessel, Nachfolger);
    END (* IF *);
END BBAumloesche;

```

Die in BBAumloesche aufgerufene Prozedur Loesche entfernt Schluessel bzw. NachfolgerSchluessel incl. seines rechts daneben stehenden Zeigers (der NIL ist). Falls nun weniger als m Schlüssel auf der Seite stehen, dann gibt es zwei Fälle:

- falls der oder die Bruderknoten *nur* m Schlüssel besitzen, dann wird ein Bruderknoten mit der Seite unter Hinzunahme des entsprechenden Schlüssels des Vaterknotens verschmolzen und der Vaterknoten wird (rekursiv) weiter darauf geprüft, ob er mindestens m Schlüssel besitzt;
- falls ein Bruderknoten *mindestens* $m+1$ Schlüssel enthält, dann wird - unter Beteiligung des Vaterknotens - ein Schlüssel zur Seite hinzugefügt und ein Schlüssel beim Bruderknoten entfernt.

2-3-Bäume

Als Spezialfall von B-Bäumen werden häufig B-Bäume der Ordnung 1, sogenannte 2-3-Bäume betrachtet. Ein B-Baum der Ordnung 1 heißt *2-3-Baum* (d.h. u.a., daß jeder innere Knoten mindestens 1 und höchstens 2 Schlüssel besitzt).

Ein *Beispiel* zeigt Abb. 6.46.

Abb. 6.46 Beispiel eines 2-3-Baumes

2-3-Bäume wachsen (durch Aufspaltung bei Überlauf) wie B-Bäume von den Blättern zur Wurzel.

Es gilt: Ein 2-3-Baum der Höhe k besitzt mindestens 2^{k-1} und höchstens 3^{k-1} Blätter.

Die Darstellung von 2-3-Bäumen kann auch durch *Binärbäume* erfolgen, deren Knoten ein *Zusatz-Bit* enthalten, um anzugeben, ob der nach rechts führende Pfeil zu einem Sohn führt oder nicht (falls drei Söhne in der nicht-binären Form vorliegen, übernimmt ein Bruder zwei Söhne!). Ein passender Datentyp läßt sich wie folgt definieren:

```

TYPE Knoten_23 = RECORD
    key          : T;
    inhalt       : ... ;
    left, right  : POINTER TO Knoten_23;
    horizontal   : BOOLEAN;
END;
```

Das obige Beispiel eines 2-3-Baumes in Binärbaumdarstellung zeigt Abb. 6.47.

Abb. 6.47 Binärbaum-Darstellung eines 2-3-Baumes

6.4.1.5 Weitere balancierte Suchbäume

In diesem Abschnitt sollen noch zwei weitere *balancierte Bäume*, HB-Bäume und gewichtsbalancierte Bäume, die für die Programmierpraxis von Interesse sind, kurz vorgestellt werden.

HB-Bäume

Ein Suchbaum B heißt *HB-Baum* $:\Leftrightarrow$

- Jeder innere Knoten von B besitzt *einen oder zwei Söhne*.
- Jeder Knoten mit nur einem Sohn besitzt einen *Bruder* mit zwei Söhnen (impliziert: Wurzel ist ein Blatt oder hat zwei Söhne).
- Alle *Blätter* von B besitzen den *gleichen Level*.

Schlüssel liegen bei HB-Bäumen redundant sowohl in den Blättern (dort auch zusätzlich die eigentlich interessierenden Datensätze), als auch auf dem Weg dorthin in inneren Knoten. Ein *Beispiel* zeigt Abb. 6.48.

Abb. 6.48 Beispiel eines HB-Baums

Den Zugriff auf ein Blatt mit dem Schlüssel x (vom Typ T) in einem HB-Baum veranschaulicht nachfolgender Algorithmus in Pseudocode.

Algorithmusentwurf:

```

(* Programm P6.13 *)
VAR Knoten, Anker : POINTER TO KnotenTyp;
    Schluessel      : SchluesselTyp;
...
BEGIN
    Knoten := Anker;
    WHILE "Knoten zeigt nicht auf ein Blatt" DO
        IF "Knoten hat genau einen Sohn" THEN
            Knoten := Knoten^.Sohn;
        ELSIF Schluessel <= Knoten^.inhalt THEN
            Knoten := Knoten^.left;
        ELSE
            Knoten := Knoten^.right;
        END (* IF *);
    END (* WHILE *);
    IF Schluessel = Knoten^.inhalt THEN
        "Erfolgreiche Suche"
    ELSE
        "Suche endete erfolglos bei dem kleinsten Blatt,
        dessen Inhalt > Schluessel ist."
    END (* IF *);
END;

```

Gewichtsbalancierte Bäume

Im Gegensatz zu höhenbalancierten Bäumen (mit Tiefenbeschränkung) wie AVL- oder B-Bäume zielen *gewichtsbalancierte Bäume* auf eine *Kardinalitätsbeschränkung* der Menge von Knoten, die jeweils in linken und rechten Teilbäumen von Knoten binärer Suchbäume zugelassen sind.

$B = (V, E)$ sei ein binärer Suchbaum mit Wurzel w und daran hängenden Teilbäumen B_l und B_r . Die *Wurzelbalance* von w in B ist

$$\beta(w) = \frac{|V_{B_l}|+1}{|V_B|+1} = \frac{|V_{B_l}|+1}{|V_{B_l}|+1 + |V_{B_r}|+1}.$$

B heißt *gewichtsbalancierter Baum*, oder auch B ist von beschränkter Balance $\alpha \Leftrightarrow$ es existiert ein $\alpha > 0$, so daß für alle $x \in V_B$ gilt: $\alpha \leq \beta(x) \leq 1-\alpha$.

Anschaulich: Für $\beta(x)$ nahe bei 0 können relativ große (bezüglich Knotenmenge!) *rechte* Teilbäume, für $\beta(x)$ nahe bei 1 relativ große *linke* Teilbäume entstehen. Für $\beta(x)$ nahe bei 1/2 sind relativ gewichtsbalancierte, d.h. gleichgroße Teilbäume zu beobachten.

Ein *Beispiel* (an den Knoten steht jeweils der entsprechende Wert für $|V_B|$) zeigt Abb. 6.49.

Abb. 6.49 Beispiel eines gewichtsbalancierten Baumes

Dieser Baum ist von beschränkter Balance $\frac{1}{4}$, weil für alle x gilt: $\frac{1}{4} \leq \beta(x) \leq \frac{3}{4}$, was Abb. 6.50 deutlich macht.

x	a	b	c	d	e	f	g	h	i	k
$\beta(x)$	7/21	3/7	1/3	1/2	2/4	1/2	1/2	4/14	1/4	2/3

x	l	m	n	o	p	q	r	s	t	u
$\beta(x)$	1/2	3/10	1/3	2/7	1/2	1/2	3/5	1/3	1/2	1/2

Abb. 6.50 Beschränkte Gewichtsbalance für Baum aus Abb. 6.49

Interessant sind Untersuchungen zum Zusammenhang von AVL-Bäumen (als höhenbalancierte Bäume) und gewichtsbalancierten Bäumen.

Die Einfüge- und Löschooperationen auf gewichtsbalancierten Bäumen sind denen auf AVL-Bäumen vergleichbar, wobei statt des Höhenbalancefaktors hier jeweils die beschränkte Gewichtsbalance zu prüfen ist und ggf. *Rotationen* notwendig werden.

6.4.2 Hashing

6.4.2.1 Begriffsbildungen und Anforderungen

Die in 6.3 und 6.4.1 vorgestellten Suchverfahren beruhen ausschließlich auf *binären* (oder höchstens *ternären*) Entscheiden. Da die Lokalisierung eines Schlüssels unter n Schlüsseln mit binären Entscheiden im Mittel mindestens $\lg(n)$ Entscheidungen benötigt, können solche Suchverfahren auch nicht besser als $O(\lg n)$ sein.

Im täglichen Leben arbeiten wir geschickter: Die Telefonnummer einer Person mit dem Namen "Zywiets" suchen wir nicht, indem wir das Telefonbuch in der Mitte aufschlagen, sondern vom Ende her, weil wir aufgrund des ersten Buchstabens dessen Ort recht genau abschätzen können.

Dies ist auch das Prinzip der *Schlüsseltransformation*: Die Idee besteht darin, aus dem Schlüssel *direkt* die Position in der Tabelle zu bestimmen. Auf diese Weise gelingt es, eine Suche mit annähernd konstantem Aufwand zu realisieren.

Suchverfahren durch Schlüsselvergleiche in Suchbäumen ermitteln zu jedem Schlüssel eine Adresse, unter der der Schlüssel abgelegt wurde bzw. melden eine erfolglose Suche. Diese Abbildung "Schlüssel $\rightarrow$ Adressen" ist *injektiv*.

Wir betrachten nun eine Abbildung $h: S \rightarrow A$ von der Schlüsselmenge S in die Adressenmenge A (oft auch als Indexbereich bezeichnet), die nicht injektiv sein muß, aber die Schlüssel *möglichst gleichmäßig* über A verteilt. Eine solche Funktion h heißt *Hash-Funktion*, auch als *Hashing* oder *Schlüsseltransformation* bezeichnet.

Probleme:

- Die Funktion h muß nicht injektiv sein, d.h. es treten Schlüssel $k_1 \neq k_2$ mit $h(k_1) = h(k_2)$ auf (*Kollision*). Es sind Strategien zur *Auflösung* solcher Konflikte zu erarbeiten.
- Es sind *effiziente* Verfahren zum Einfügen und Löschen zu entwickeln.
- Die Zahl der einzufügenden Schlüssel ist vorher meist nicht bekannt. Man muß daher die Möglichkeit haben, A zu vergrößern (*dynamisches Hashing*).

Beispiel

Eine Folge von drei Buchstaben ist darauf zu testen, ob sie Abkürzung eines Monatsnamens JAN, FEB, MAE, APR, MAI, JUN, JUL, AUG, SEP, OKT, NOV, DEZ ist. Möglichkeiten:

1. Abspeicherung in einem *optimalen Suchbaum*.
2. Man nimmt eine *andere Baumstruktur*.
3. Man sucht eine *Hash-Funktion* $h: \{\text{JAN, FEB, ..., DEZ}\} \rightarrow \{0, 1, \dots, m\}$ (mit $m \geq 11$), die leicht zu berechnen ist, und verwendet eine Tabelle mit Indizierung 0, 1, ..., m. Diese Tabelle stellt den nach Indizes geordneten Zielbereich dar und wird als *Hash-Tabelle* bezeichnet.

Als geschickte Hash-Funktion kann man z.B. für die Abkürzungen der Monatsnamen wählen (nr sei die "Nummer" eines Buchstabens im Alphabet):

$h: (\text{Char})^3 \rightarrow 0..14$ mit

$$h(x_1x_2x_3) = (7 \text{ nr}(x_1) + 5 \text{ nr}(x_2) + 2 \text{ nr}(x_3)) \bmod 15.$$

Für die gegebene Schlüsselmenge JAN, FEB, ..., DEZ ist h sogar eine injektive Abbildung (man spricht in diesem Fall auch von einem *perfekten Hashing*), wie die *Wertetabelle* (Auflistung der Werte für alle Schlüssel) in Abb. 6.51 zeigt.

$x_1 x_2 x_3$	= JAN FEB MAE APR MAI JUN JUL AUG SEP OKT NOV DEZ											
$h(x_1 x_2 x_3)$	13	11	1	3	9	8	4	6	10	5	7	0

Abb. 6.51 Wertetabelle (Perfektes Hashing von Monatsabkürzungen)

Dieses Beispiel bezieht sich auf eine *statische Schlüsselmenge*, d.h. die Schlüssel ändern sich nicht mehr durch Einfügen oder Löschen, wohl aber die eigentlichen Daten (z.B. meteorologische Informationen zu den Monaten). Die entsprechende Hash-Tabelle hat 14 "Plätze" (Indizierung von 0 bis 13), von denen drei (Indizes 2, 12 und 14) nicht belegt sind

Wenn man nun zu einem Monatsnamen $x_1x_2x_3$ Daten sucht oder ändern will, dann berechnet man $h(x_1x_2x_3)$ und hat direkt die richtige Stelle/Adresse in der Hash-Tabelle ermittelt. Man hat also nur *eine* Berechnung durchzuführen, nimmt aber dafür einen Speicherplatzverlust für die nicht belegten Plätze der Hash-Tabelle in Kauf.

Wir werden sehen, daß man die Hash-Technik auch für sich *dynamisch* ändernde Schlüsselbestände verwenden kann.

6.4.2.2 Perfektes Hashing

Seien einhundert Tabelleneinträge unter den Nummern 1 bis 100 zu verwalten. In diesem Fall ist die Lösung trivial: Wir verwenden die Nummer (d.h. den Schlüssel) als Index und können jedes Element direkt ansprechen.

In der Praxis bilden aber die Schlüssel meist nicht wie oben angenommen einen geeigneten Index, und zwar aus zwei Gründen:

- Es handelt sich fast nie um Zahlen, sondern um Zeichenketten (z.B. Namen, Bezeichner eines Programms, die von einem Übersetzer verarbeitet werden sollen).
- Die Menge der möglichen Schlüssel ist sehr groß, die tatsächlich verwendeten stellen nur einen winzigen Bruchteil dar.

Beispiel: Seien Bezeichner aus 16 Zeichen zugelassen, bestehend aus Buchstaben und Ziffern. Werden Groß- und Kleinbuchstaben nicht unterschieden (wie z.B. in Pascal) und ist das erste Zeichen ein Buchstabe, so gibt es dennoch allein $26 \cdot 36^{15}$ verschiedene Bezeichner mit 16 Zeichen, also mehr als 10^{24} . Eine "Übersetzungseinheit" in einem Programm (z.B. ein Modul) enthält höchstens 10^2 Bezeichner.

Aus diesen Gründen kann der Schlüssel nicht direkt verwendet werden, er wird zunächst in ein Intervall des Basistyps INTEGER (oder CARDINAL) transformiert. Dieses Intervall sollte groß genug, aber nicht unsinnig groß sein, um eine Indizierung mit dem so ermittelten *Primärindex* (der eigentlich bereits sekundär ist) zu erlauben.

Beispiel: Daten von 10 Personen sollen in einer Tabelle abgelegt werden. Die Personen sind: *Agnes, Annegret, Arthur, Beat, Boris, Cäsar, Clemens, Dagmar, Dagobert, Dolores*.

Als Hash-Funktion dient:

```
PROCEDURE h ( Name : ARRAY OF CHAR ) : CARDINAL;
BEGIN (* h *)
  RETURN 3 * (ORD (Name [0]) - ORD ("A"))
    + (ORD (Name [HIGH (Name)]) - ORD ("r"));
END h;
```

Damit lassen sich die zehn genannten Namen in das Intervall 0 (*Arthur*) bis 11 (*Dagobert*) abbilden.

Das für die Schlüsseltransformation vorgenommene "Zerhacken" ("hashing") der Bezeichner hat dem Verfahren den Namen gegeben.

Am vorangehenden Beispiel ist bemerkenswert, daß wieder alle Schlüssel durch die Schlüsseltransformation auf verschiedene Primärindizes abgebildet werden, d.h. *h* ist injektiv und damit perfekt.

Offenbar hat das *perfekte Hashing* zwei notwendige Voraussetzungen:

- Die Menge der Schlüssel muß a priori bekannt (und damit konstant) sein.
- Es darf höchstens so viele Schlüssel geben, wie die Tabelle (d.h. der Primärindex) Raum bietet, aber die Tabelle darf nicht viel größer sein, da sonst eine zu große Streuung und ein zu hoher Speicherbedarf entstehen.

Da in den meisten Anwendungen die erste Bedingung nicht (und die zweite nicht sicher) erfüllt ist, sind die Anwendungsmöglichkeiten für das perfekte Hashing sehr begrenzt.

6.4.2.3 Kollisionsbehandlung

In der Praxis sind die Voraussetzungen für perfektes Hashing meist nicht gegeben, und wir müssen damit rechnen, daß unterschiedliche Schlüssel auf gleiche Primärindizes abgebildet werden, d.h. *Kollisionen* $h(s_1) = h(s_2)$ bei $s_1 \neq s_2$ eintreten.

Es genügt nun nicht mehr, nur die zu s gehörige Information in der Tabelle unter dem Index $h(s)$ abzulegen. Erstens muß auch s abgelegt werden, damit im Falle einer Kollision festgestellt werden kann, um welchen Schlüssel es sich handelt. Zweitens muß eine Strategie für die *Kollisionsauflösung* entwickelt werden.

Natürlich wird man versuchen, die Hash-Funktion so zu wählen, daß Kollisionen selten sind; ein zweites Kriterium ist die effiziente Ausführung. Betrachten wir dazu die beiden *Extreme*:

- Die Hash-Funktion $h(s) = 1$ ist sehr effizient, erzeugt aber leider stets (d.h. vom zweiten Eintrag an) Kollisionen und ist daher völlig ungeeignet.
- Eine Gewichtung der einzelnen Zeichen mit *Primzahlen* (bei einem Schlüssel aus den Zeichen z_0 bis z_k):

$$2 \cdot \text{ORD}(z_0) + 3 \cdot \text{ORD}(z_1) + 5 \cdot \text{ORD}(z_2) + 7 \cdot \text{ORD}(z_3) + \dots$$

gefolgt von einer Abbildung mit Modulo-Funktion in den Indexbereich ist hinsichtlich Kollisionen sehr wirksam, aber zu rechenintensiv.

Typisch sind folgende *Kompromißlösungen*:

- Verwendung des gesamten Bezeichners als Zahl, Abbildung mit MOD in den Indexbereich.

```

h := ORD (z1)
FOR i := 2 TO k DO
  h := (256 * h + ORD (zi)) MOD n; (* n ist Tabellenlänge *)
END;
```

Die Multiplikation wird als Shift-Operation effizient implementiert. Dabei ist unterstellt, daß die Zeichen in 8 bit gespeichert sind. Man beachte, daß n und 256 teiler-

fremd sein sollten; ist n im ungünstigsten Fall gleich 256, so ist der Aufwand nutzlos, und es ergibt sich: $h = \text{ORD}(z_k)$.

- Verwendung einzelner Zeichen des Bezeichners, eventuell zusätzlich der Länge (hier k genannt).

Beispiel: $h := (7 \cdot \text{ORD}(z_0) + 11 \cdot \text{ORD}(z_{k-1}) + 19 \cdot k) \text{ MOD } n$.
Auch hier sollten n und die vorkommenden Konstanten (7,11,19) relativ prim (teilerfremd) sein.

Allgemein ist es eine sichere Regel, als Länge der Tabelle eine Primzahl zu wählen. Um die Zahl der Kollisionen gering zu halten, empfiehlt es sich, die Tabelle mindestens so groß zu machen, daß sie nie mehr als 80% gefüllt ist. Außerdem ist zu beachten, daß bei der Wahl der Hash-Funktion bestehende Namenskonventionen nach Möglichkeit zu berücksichtigen sind (z.B. ist es sehr ungünstig, wenn die ersten drei Zeichen eines Bezeichners das Modul bezeichnen und gerade die ersten drei Zeichen zum Hashing herangezogen werden).

Eine einfache Lösung des Kollisionsproblems besteht darin, den Überlauf in die *Halde* zu verdrängen, d.h. eine *externe Kollisionsbehandlung* durchzuführen.

Die Hash-Tabelle wird dabei wie folgt aufgebaut:

```
CONST n = .....;                (* Tabellenlaenge, Primzahl! *)
TYPE NachfolgerZeiger = POINTER TO EintragTyp;
EintragTyp = RECORD
    CASE Belegt : BOOLEAN OF
    | FALSE : (*leer*)
    | TRUE  : Schlüssel : SchluesselTyp;
              Inhalt    : InhaltTyp;
              Nachfolger : NachfolgerZeiger
    END (*CASE*);
END (*RECORD*);
VAR HashTabelle : ARRAY [0..n-1] OF EintragTyp;
```

Natürlich müssen die Elemente der Tabelle mit `Belegt := FALSE` initialisiert werden. Wird ein Element besetzt, so wird der Nachfolger auf `NIL` gesetzt; finden später Kollisionen statt, so können beliebig viele weitere Elemente auf der *Halde* alloziert und durch einfache Verkettung angehängt werden.

Da aber in der Praxis nicht einzelne Zahlen, sondern umfangreichere Information zu speichern ist, empfiehlt es sich, *alle* Einträge auf der Halde abzulegen. Die Tabelle im *Keller* wird dadurch wesentlich kleiner (sie enthält nur noch Zeiger) und kann ohne erhebliche Nachteile reichlich groß bemessen werden (weniger Kollisionen):

```
VAR HashTabelle : ARRAY [0..n-1] OF NachfolgerZeiger;
```

In diesem Fall werden die Elemente der Tabelle mit `NIL` initialisiert, und man benötigt keinen varianten Record mehr (Belegt entfällt).

Soll (oder kann) die Halde für die Kollisionsauflösung nicht in Anspruch genommen werden, können die Überläufe auch in der Tabelle selbst gespeichert werden: Sie belegen dann Plätze, die bislang freigeblieben sind, was zu einer *internen Kollisionsbehandlung* führt.

In diesem Fall muß eine Strategie festgelegt werden, wie vom Ort der Kollision aus (reproduzierbar!) ein freier Platz gefunden wird. Ferner muß die Möglichkeit, daß die Tabelle insgesamt gefüllt wird, in die Überlegung einbezogen werden. Dafür wird die Berechnung einer sogenannten *Sondierfolge* definiert. Die Elemente dieser Folge legen, beginnend mit dem Primärindex eines Schlüssels, die Indizes fest, an denen nach einem freien Platz gesucht werden soll. Man bezeichnet diese Suche als *Sondieren*.

Lineares Sondieren

Das einfachste Verfahren zum Sondieren besteht darin, vom Kollisionsort sequentiell (und am Tabellenende zyklisch) weiterzusuchen, bis ein freier Platz oder der Ausgangspunkt erreicht ist. Statt des zweiten Kriteriums kann auch die Zahl der Einträge überwacht werden. Die Sondierfolge $h_i(s)$ wird durch

$$h_0(s) = h(s)$$

$$h_i(s) = (h_0(s) + i) \text{ MOD } n, \quad i = 1, \dots, n-1$$

berechnet.

Auch beim Sondieren kann es natürlich zu weiteren Kollisionen (*Sekundärkollisionen*) kommen. Es ist daher vorteilhaft, beim Sondieren für verschiedene Schlüssel *unterschiedliche* Schrittweiten zu wählen. Man legt dazu - ähnlich der Hash-Funktion $h(s)$ - eine Funktion $d(s)$ fest, etwa die Länge von s , und sucht mit dieser Schrittweite nach einem freien Platz. Beim linearen Sondieren ist $d(s)$ jedoch meistens eine vom Sondierfolgenindex i unabhängige Funktion. Die Sondierfolge $h_i(s)$ wird wie folgt berechnet:

$$h_0(s) = h(s)$$

$$h_i(s) = (h_0(s) + i \cdot d(s)) \text{ MOD } n, \quad i = 1, \dots, n-1.$$

Algorithmus:

(* Programm P6.14 *)

```

IF ZahlEintraege < n THEN
  Index := h (s);
  Delta := d (s);
  LOOP
    IF NOT HashTabelle[Index].Belegt THEN (* freier Platz *)
      Eintragen; INC (ZahlEintraege); EXIT
    ELSIF HashTabelle[Index].Schluessel = s THEN
      (* s bereits vorhanden, eventuell Meldung *)
      EXIT;
    ELSE
      Index := (Index + Delta) MOD n;

```



```

        END (* IF *);
    END (* LOOP *);
ELSE
    (* Meldung 'Tabelle voll' *)
END (* IF *)

```

Auch hier ist eine Primzahl als Länge n der Tabelle vorteilhaft; andernfalls (wenn d und n nicht teilerfremd sind) wird nur ein Teil der Tabelle durchgemustert, so daß der angegebene Algorithmus nicht sicher terminiert.

Beispiel: $d=8$ und $n=100$ führt nach $\frac{200}{8}$ Schritten auf einen Zyklus, es wird also nur ein Viertel der Tabelle geprüft.

Der Nutzen dieses differenzierten Sondierens mit schlüsselabhängiger Schrittweite ist zweifelhaft, da man weiterhin evtl. mit solchen Einträgen kollidiert, die einen um Vielfache von d verschiedenen Primärindex haben.

Quadratisches Sondieren

Beim quadratischen Sondieren wird darum das d von Schritt zu Schritt modifiziert, und zwar durch folgende Berechnung: $d_1 = 1$, $d_{i+1} = d_i + 2$.

Auf diese Weise durchläuft die Distanz zum Primärindex gerade die Quadratzahlen,

wie sich durch Induktion zeigen läßt: $\sum_{i=1}^k d_i = k^2$.

Für die Berechnung der *Sondierfolge* gilt dann:

$$\begin{aligned}
 h_0(s) &= h(s); \\
 h_i(s) &= (h_0(s) + d_i) \text{ MOD } n = (h_0(s) + i^2) \text{ MOD } n
 \end{aligned}$$

Algorithmusentwurf:

```

                                                                    (* Programm P6.15 *)
Index := h(s); Delta := 1;
LOOP
    IF NOT HashTabelle[Index].Belegt THEN
        Eintragen;
        EXIT;
    ELSIF HashTabelle[Index].Schluessel = s THEN
        (* s bereits vorhanden *)
        EXIT;
    ELSIF Delta > n THEN
        (* Abbruch, da kein Platz gefunden *)
        EXIT;
    ELSE
        Index := (Index + Delta) MOD n;
        INC (Delta, 2);
    END (* IF *);
END (* LOOP *);

```


Natürlich ist auch hier zu klären, wie weit die Tabelle durchgemustert wird.

Offenbar können nicht mehr als $\frac{n}{2}$ Elemente durchgemustert werden, denn es gilt:

$$i^2 \bmod n = (n-i)^2 \bmod n = (n+i)^2 \bmod n.$$

Darauf beruht das Abbruchkriterium $\Delta > n$. Optimal ist also der Fall, in dem alle Werte $0 \leq i < \frac{n}{2}$ auf verschiedene Indizes fallen.

Trete die erste Wiederholung beim k -ten Sondieren auf (ausgehend vom Primärindex $p=h(s)$):

$$(p + k^2) \bmod n = (p + j^2) \bmod n \text{ mit } k > j \geq 0, k \text{ minimal}$$

$$\Rightarrow k^2 - j^2 = m \cdot n \text{ mit } m \in \mathbb{N}$$

$$\Leftrightarrow (k - j)(k + j) = m \cdot n.$$

Ist n eine Primzahl, so ist die kleinste Lösung für k :

$$k + j = n \wedge k - j = m = 1$$

$$\Rightarrow k = \frac{n+1}{2} > \frac{n}{2} \text{ wie angestrebt.}$$

Ist n dagegen keine Primzahl, so gibt es kleinere Lösungen für k , d.h. bestimmte Felder werden u.U. mehrfach geprüft.

Vergleich der Sondiervverfahren

Beim Sondieren werden drei Ziele angestrebt:

- (a) einfache (effiziente) Implementierung
- (b) möglichst wenige Sekundärkollisionen
- (c) möglichst vollständige Ausnutzung der Tabelle.

Das einfache Verfahren ist optimal im Sinne von (a), das quadratische Sondieren ist ebenfalls sehr effizient, das differenzierte Sondieren kann nur durch eine komplizierte Funktion $d(s)$ belastet sein.

Die "Entzerrung" der Sekundärindizes (Kriterium (b)) gelingt mit dem simplen Ansatz nicht; fallen viele Primärindizes in einen bestimmten Bereich der Hash-Tabelle, so wird dieser lückenlos gefüllt und erfordert dann entsprechend viele Sondierschritte. Beim quadratischen Sondieren können sich solche Bereiche nicht bilden.

Die Durchmusterung der Tabelle ist bei den beiden linearen Verfahren vollständig (bei Primzahl n), beim quadratischen Sondieren ist sie auf 50% begrenzt. Es zeigt sich auch hier wieder, daß simple Algorithmen oft besser abschneiden, als man zunächst vermutet; da ihre Einfachheit auch die Realisierung und Wartung vereinfacht, sind sie stets vorzuziehen, bis die Notwendigkeit komplizierterer Algorithmen durch Messungen nachgewiesen ist.

6.4.2.4 Löschen in Hash-Tabellen

Sollen einzelne Einträge aus einer Hash-Tabelle gelöscht werden, so erscheint das Vorgehen zunächst klar: Der Eintrag wird gesucht und gelöscht. Bei externer Kollisionsbehandlung ist damit die Entfernung aus der Kette auf der Halde verbunden, bei interner liegt der Fall komplizierter.

Jeder Eintrag kann in der Sondierkette weiterer Einträge liegen, die nicht auf ihrem Primärindex gespeichert werden konnten. Wird der Eintrag entfernt, so ist die Sondierkette unterbrochen und die folgenden Einträge sind nicht mehr auffindbar.

Darum ist eine der beiden folgenden Maßnahmen erforderlich:

- (a) Die *Tabelle wird reorganisiert*, d.h. alle in Frage kommenden anderen Einträge werden geprüft und wenn nötig verschoben.
- (b) Ein gelöschter Eintrag erhält den speziellen *Status "gelöscht"*. Er kann dann zwar wieder belegt werden, unterbricht aber Sondierketten nicht.

Die Lösung nach (a) kommt praktisch nur dann in Frage, wenn linear mit Schrittweite 1 sondiert worden war. In diesem Fall genügt es, bis zur nächsten Lücke zu suchen. Die Lösung nach (b) hat den Nachteil, daß Sondierketten länger als nötig werden; sie ist daher nur dann geeignet, wenn Löschen selten vorkommt.

6.4.2.5 Aufwandsabschätzung

Eine Aufwandsabschätzung für die *interne Kollisionsauflösung* ist nur unter idealisierenden Annahmen möglich: Die Wahrscheinlichkeit einer Primär- oder Sekundärkollision soll nur vom *Füllungsgrad* der Tabelle abhängen. Dadurch sind Effekte, wie sie bei der internen Kollisionsbehandlung diskutiert wurden, ausgeschlossen. Man muß daher auch das Ergebnis mit entsprechenden Vorbehalten würdigen.

Sei die Tabellenlänge n und seien in der Tabelle bereits k Einträge, dann ist die *Wahrscheinlichkeit einer Primärkollision* $q_1 = \frac{k}{n}$.

Die einer Primär- und j weiterer *Sekundärkollisionen* ist

$$q_{j+1} = \frac{k}{n} \cdot \frac{k-1}{n-1} \cdot \dots \cdot \frac{k-j}{n-j} = \frac{k!}{(k-j-1)!} \cdot \frac{(n-j-1)!}{n!}.$$

Jetzt sind von den n Plätzen $j+1$ getestet, also kommen für die nächste Sondierung noch $n-(j+1)$ in Frage, von denen $n-k$ frei sind. Die Wahrscheinlichkeit, gerade beim $j+2$ -ten Sondierschritt, also nach $j+1$ Kollisionen, einen freien Platz zu finden, ist demnach

$$p_{j+2} = q_{j+1} \cdot \frac{n-k}{n-j-1} \quad \text{oder nach Substitution mit } i \text{ für } j+2:$$

$$p_i = q_{i-1} \cdot \frac{n-k}{n-i+1} = \frac{k!}{(k-i+1)!} \cdot \frac{(n-i+1)!}{n!} \cdot \frac{n-k}{n-i+1}$$

Durch Kürzen, Umordnen und Erweitern folgt:

$$p_i = \frac{k! \cdot (n-k)!}{n!} \cdot \frac{(n-i)!}{(n-k-1)! \cdot (k-i+1)!} = \binom{n}{k}^{-1} \binom{n-i}{n-k-1}$$

in der Notation für Binomialkoeffizienten.

Der *Erwartungswert* für die Zahl der Sondierungen beim Eintrag des $k+1$ -ten Elements ist:

$$E_{k+1} = \sum_{i=1}^{k+1} i p_i = \sum_{i=1}^{k+1} \sum_{j=1}^i p_i$$

$$p_i = \sum_{j=1}^{k+1} \sum_{i=j}^{k+1} p_i = \binom{n}{k}^{-1} \sum_{j=1}^{k+1} \sum_{i=j}^{k+1} \binom{n-i}{n-k-1}$$

Den Übergang zur zweiten Doppelsumme kann man nachvollziehen, indem man die Doppelsumme in Dreiecksform zeilenweise aufschreibt und dann spaltenweise zusammenfaßt. Durch weitere arithmetische Umformungen unter Ausnutzung des *Pascal'schen Dreiecks* erhält man schließlich:

$$E_{k+1} = \frac{n+1}{n-k+1}.$$

Der mittlere Aufwand für das Eintragen eines von m Schlüsseln in die Hash-Tabelle ergibt sich damit zu

$$E = \frac{1}{m} \sum_{k=1}^m E_k = \frac{1}{m} \sum_{k=1}^m \frac{n+1}{n-k+2} = \frac{n+1}{m} \sum_{j=n-m+2}^{n+1} \frac{1}{j}$$

$$= \frac{n+1}{m} \left(\sum_{j=1}^{n+1} \frac{1}{j} - \sum_{j=1}^{n-m+1} \frac{1}{j} \right) = \frac{n+1}{m} (H_{n+1} - H_{n-m+1})$$

worin H_i für die Summe der harmonischen Reihe bis zum Glied i steht. Da sich diese Summe durch den natürlichen Logarithmus annähern läßt, folgt:

$$H_{n+1} - H_{n-m+1} \approx \ln(n+1) - \ln(n-m+1) = \ln \frac{n+1}{n+1-m}.$$

Damit ergibt sich näherungsweise

$$E_i \approx \frac{n+1}{m} \ln \frac{n+1}{n+1-m} = -\frac{1}{a} \ln(1-a) \text{ mit } a = \frac{m}{n+1} \approx \frac{m}{n}.$$

Abb. 6.52 Mittlerer Sondieraufwand bei interner Kollisionsauflösung

Folglich ist der Sondieraufwand nur vom Füllungsgrad $\frac{m}{n}$ abhängig, nicht vom absoluten Wert von m oder n . Wir haben in diesem Sinne also *konstanten Aufwand*!

6.4.2.6 Implementierung von Kollisionsauflösungen

Zum Abschluß von 6.4.2 folgt noch je eine Implementierung für interne und externe *Kollisionsauflösung* in Form einer Tabellenkapsel. Die Funktionalität ist in beiden Fällen völlig gleich, wenn man davon absieht, daß bei interner Kollisionsauflösung nur soviele Einträge möglich sind, wie die Tabelle Plätze hat; dafür gibt es die Probleme der Halden-Verwaltung nicht.

```

DEFINITION MODULE TabKap;                                     (* Programm P6.16a *)
TYPE Zeichenkette = ARRAY [0..20] OF CHAR;
   SchluesselTyp = RECORD
       String: Zeichenkette;
       Laenge: INTEGER;
   END (* RECORD *);
   InhaltTyp    = REAL; (* Nicht sehr sinnvoll: In der Praxis
                           koennte man hier z.B. die Daten einer
                           Person, die Semantik eines reserv.
                           Worts etc. speichern *)

```

```

        ElementTyp      = RECORD
            Schluessel : SchluesselTyp;
            Inhalt      : InhaltTyp;
        END (* RECORD *);

    (***** PROZEDUREN *****)
    PROCEDURE Eintragen ( (* in *) Element : ElementTyp;
                          (* out *) VAR OK  : BOOLEAN);
    (* Traegt Element (Schluessel, Inhalt) in die Tabelle ein:
       OK = TRUE : Eintrag erfolgreich durchgefuehrt.
       OK = FALSE: Tabelle ist voll oder ein Eintrag mit gleichem
                    Key ist schon vorhanden (besser: unterscheiden) *)
    (*****)
    PROCEDURE Holen ( (* in *) SuchSchluessel : SchluesselTyp;
                     (* out *) VAR Element   : ElementTyp;
                     (* out *) VAR OK        : BOOLEAN);
    (* Findet Element mit Schluessel 'SuchSchluessel':
       OK = TRUE : Element gefunden
       OK = FALSE: Element war nicht in der Liste. *)
    (*****)
    PROCEDURE Loeschen ( (* in *) LoeschSchluessel: SchluesselTyp;
                        (* out *) VAR OK          : BOOLEAN);
    (* Loescht Element mit Schluessel 'LoeschSchluessel':
       OK = TRUE : Element geloescht.
       OK = FALSE: Element war nicht in der Liste *)
    (*****)
    END TabKap.

```

Implementierung mit interner Kollisionsauflösung

Der Algorithmus beruht auf der Annahme, daß die Tabelle nur schwach gefüllt wird oder Löschen nur sehr selten vorkommt. Andernfalls führt die Markierung gelöschter Elemente zu ineffizienter Suche durch einen großen Teil der Tabelle.

```

IMPLEMENTATION MODULE TabKap;                                (* Programm P6.16b *)

    (***** Konstanten, Typen und Variablen *****)
    CONST TabLaenge      = 101;                               (* Primzahl *)
    TYPE HashIndexTyp    = [0..TabLaenge-1];
    VAR  ZahlEintraege : HashIndexTyp; (* Mindestens ein Element muss
                                         frei bleiben *)
    TYPE StatusTyp       = (Belegt, Frei, Geloescht);
    TYPE TabellenElementTyp = RECORD
        CASE Status: StatusTyp OF
            | Belegt : Eintrag: ElementTyp
            | Frei, Geloescht : (* leer *)
        END (* CASE *);
    END (* RECORD *);
    VAR Tabelle: ARRAY HashIndexTyp OF TabellenElementTyp;

```



```

(***** Interne Prozeduren *****)

(***** Suchen *****)
PROCEDURE Suchen ( (* in *) SuchSchluessel : SchluesselTyp;
                  (* out *) VAR vorhanden : BOOLEAN;
                  (* out *) VAR Index      : HashIndexTyp);
(* sucht Element mit SuchSchluessel. Falls vorhanden, zeigt Index
   nachher auf dieses Element. Falls nicht vorhanden, zeigt Index
   auf den ersten beim Suchen angetroffenen freien Platz (ist sicher
   vorhanden). *)

(***** Hash-Funktion *****)
PROCEDURE Hash ( (* in *) HashSchluessel: SchluesselTyp): INTEGER;
VAR i, Summe: INTEGER;
BEGIN
  Summe := ORD (HashSchluessel.String [0]);
  FOR i := 1 TO HashSchluessel.Laenge-1 DO
    Summe := Summe + ORD (HashSchluessel.String [i]);
  END (* FOR *);
  RETURN Summe MOD TabLaenge;
END Hash;

(***** Step *****)
PROCEDURE Step ( (* inout *) VAR Argument: HashIndexTyp);
(* muss vollstaendige Durchmusterung garantieren, sonst Fehler *)
BEGIN (* lineares Sondieren mit Schrittweite 1 *)
  Argument := (Argument + 1) MOD TabLaenge;
END Step;

(***** String-Vergleich *****)
PROCEDURE Equal
  ( (* in *) Schluessel1, Schluessel2: SchluesselTyp): BOOLEAN;
VAR i, Lng: INTEGER;
BEGIN
  WITH Schluessel1 DO
    IF Laenge # Schluessel2.Laenge THEN
      RETURN FALSE;
    ELSE
      i := 0;
      WHILE (i <= Laenge) AND
        (String [i] = Schluessel2.String [i]) DO
        INC(i);
      END;
      RETURN (i > Laenge); (* d.h. gleich auf ganzer Laenge *)
    END (* IF *);
  END (* WITH *);
END Equal;

(***** Rumpf Suchen *****)
VAR Index0, LeerIndex : HashIndexTyp; LeerGef : BOOLEAN;
(* LeerIndex zeigt erstes freies Element, wenn LeerGef *)
BEGIN (* Suchen *)
  Index0 := Hash (SuchSchluessel);          (* Primaerindex *)
  Index := Index0;                          (* laufender Index *)
  LeerGef := FALSE; vorhanden := FALSE;    (* Voreinstellung *)

```



```

LOOP
  WITH Tabelle [Index] DO
    IF Status = Frei THEN EXIT          (* leerer Platz gefunden *)
    ELSIF Status = Geloescht THEN
      IF NOT LeerGef THEN               (* erstes geloeschtes
                                         Element gefunden *)
        LeerGef := TRUE;
        LeerIndex := Index; (* Index merken *)
      END (* IF *);
      (* bleibt nur noch der Fall Status = Belegt *)
      ELSIF Equal (Eintrag.Schluessel, SuchSchluessel) THEN
        vorhanden := TRUE; EXIT        (* Schluessel gefunden *)
      END (* IF *);
    END (* WITH *);
    Step (Index);                      (* weitersondieren *)
    IF Index = Index0 THEN EXIT; END;   (* Tabelle vollstaendig
                                         durchsucht *)
  END (* LOOP *);
  IF LeerGef AND NOT vorhanden THEN Index := LeerIndex; END;
END Suchen;

(***** Exportierte Prozeduren *****)

(***** Eintragen *****)
PROCEDURE Eintragen ( (* in *) Element : ElementTyp;
                      (* out *) VAR OK : BOOLEAN);
VAR Index: HashIndexTyp;
    SchonVorh : BOOLEAN;
BEGIN
  Suchen (Element.Schluessel, SchonVorh, Index);
  OK := (NOT SchonVorh) AND
        (ZahlEintraege < TabLaenge - 1);
  IF OK THEN
    INC (ZahlEintraege);
    WITH Tabelle[Index] DO
      Status := Belegt;
      Eintrag := Element;
    END (* WITH *);
  END (* IF *);
END Eintragen;

(***** Holen *****)
PROCEDURE Holen ( (* in *) SuchSchluessel : SchluesselTyp;
                  (* out *) VAR Element : ElementTyp;
                  (* out *) VAR OK : BOOLEAN);
VAR Index: HashIndexTyp;
BEGIN
  Suchen (SuchSchluessel, OK, Index);
  IF OK THEN Element := Tabelle [Index].Eintrag; END;
END Holen;

```

```

(***** Loeschen *****)
PROCEDURE Loeschen ( (* in *) LoeschSchluessel: SchluesselTyp;
                    (* out *) VAR OK: BOOLEAN);
VAR Index: HashIndexTyp;
BEGIN
  Suchen (LoeschSchluessel, OK, Index);
  IF OK THEN
    Tabelle [Index].Status := Geloescht;
    DEC (ZahlEintraege);
  END (* IF *);
END Loeschen;

(***** Initialisierung *****)
VAR i: HashIndexTyp;
BEGIN (* TabKap *)
  FOR i := MIN (HashIndexTyp) TO MAX (HashIndexTyp) DO
    Tabelle [i].Status := Frei;
  END (* FOR *);
  ZahlEintraege := 0;
END TabKap.

```

Implementierung mit externer Kollisionsauflösung

Die folgende Implementierung verwendet Prozedur-Parameter, damit der Code für die Suche nach einem Element nicht mehrfach geschrieben werden muß. Dadurch können die Prozeduren Eintragen, Holen und Loeschen dieselbe Such-Prozedur verwenden und trotzdem nach dem Finden oder Nichtfinden *verschiedene* Reaktionen vorgeben. Man beachte, daß auch die Lösung oben so arbeitet, doch kann dort ein Index zurückgegeben werden; hier handelt es sich um Zeiger, die nicht ohne Verlust der Identität aus der Prozedur heraufgereicht werden können.

Diese Konzeption ist hier natürlich kaum angemessen, sie demonstriert aber das Prinzip der *Prozedur-Parameter*. Die als Parameter eingesetzten Prozeduren Anhaengen, Lesen und Aushaengen müssen aufgrund der Definition von MODULA-2 auf globaler Ebene vereinbart sein; logisch sind sie aber lokal zu den Prozeduren, vor denen sie jeweils stehen.

```

IMPLEMENTATION MODULE TabKap;                                (* Programm P6.16c *)

FROM SYSTEM IMPORT VAL;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

(***** Konstanten, Typen und Variablen *****)
CONST TabLaenge = 30;                                       (* Groesse der Hash-Tabelle *)
TYPE ElementZeiger = POINTER TO ListenElement;
   ListenElement = RECORD
     Eintrag : ElementTyp;
     Next    : ElementZeiger;
   END (* RECORD *);
VAR Tabelle : ARRAY [0..TabLaenge-1] OF ElementZeiger;

```



```

BEGIN
  OK := Zeiger = NIL;
  IF OK THEN
    ALLOCATE (Zeiger, SIZE (ListenElement));
    WITH Zeiger^ DO Eintrag := Element; Next := NIL END;
  END (* IF *);
END Anhaengen;

PROCEDURE Eintragen ( (* in *) Element : ElementTyp;
                      (* out *) VAR OK   : BOOLEAN);
BEGIN
  Suchen (Element.Schluessel, Anhaengen, Element, OK);
END Eintragen;

(***** Holen *****)
PROCEDURE Lesen ((* inout *) VAR Zeiger : ElementZeiger;
                 (* out *) VAR Element : ElementTyp;
                 (* out *) VAR OK      : BOOLEAN);
BEGIN
  OK := Zeiger # NIL;
  IF OK THEN Element := Zeiger^.Eintrag END;
END Lesen;

PROCEDURE Holen ( (* in *) SuchSchluessel : SchluesselTyp;
                  (* out *) VAR Element   : ElementTyp;
                  (* out *) VAR OK        : BOOLEAN);
BEGIN
  Suchen (SuchSchluessel, Lesen, Element, OK);
END Holen;

(***** Loeschen *****)
PROCEDURE Aushaengen ((* inout *) VAR Zeiger : ElementZeiger;
                      (* inout *) VAR Dummy  : ElementTyp;
                      (* out *) VAR OK       : BOOLEAN);
VAR Hilf: ElementZeiger;
BEGIN
  OK := Zeiger # NIL;
  IF OK THEN
    Hilf := Zeiger; Zeiger := Zeiger^.Next;
    DEALLOCATE (Hilf, SIZE (ElementTyp));
  END (* IF *);
END Aushaengen;

PROCEDURE Loeschen ( (* in *) SuchSchluessel : SchluesselTyp;
                     (* out *) VAR OK        : BOOLEAN);
VAR Dummy: ElementTyp;
BEGIN
  Suchen (SuchSchluessel, Aushaengen, Dummy, OK);
END Loeschen;

```



```
(***** Initialisierung *****)  
VAR i: INTEGER;  
BEGIN  
  FOR i := 0 TO TabLaenge - 1 DO  
    Tabelle [i] := NIL;  
  END (* FOR *);  
END TabKap.
```

6.5 Sortieren

6.5.1 Klassifizierung und allgemeine Betrachtungen

Sortieren ist häufig eine wichtige elementare Aufgabe innerhalb komplexer Algorithmen. Das Sortieren dient i.a. dem schnelleren Wiederfinden von Informationen. Beispiele für sortierte Informationsmengen sind Lexika, Tabellen, Adressenbücher, Kataloge usw.

Man unterscheidet

- *internes Sortieren* : die zu sortierende Folge befindet sich im *Hauptspeicher* (wahlfreier Zugriff);
- *externes Sortieren* : die zu sortierende Folge ist auf externe Speichermedien wie Bänder oder Platten ausgelagert (i.a. sequentieller Zugriff).

Verglichen mit großen Anwendungsprogrammen sind Sortieralgorithmen relativ kurz, dafür aber recht kompliziert, und ihre Effizienz ist von großer Bedeutung.

Am Beispiel des Sortierens kann man viele Aspekte, die beim Entwickeln von Algorithmen zu berücksichtigen sind, erläutern, z.B.:

- Probleme haben in der Regel sehr *vielen* brauchbare Lösungen (es werden deshalb mehrere Algorithmen zur Lösung des Sortierproblems vorgestellt);
- aus unterschiedlichen Ideen entwickeln sich unterschiedliche Algorithmen ("*top-down - Prinzip*");
- die Aufwandsanalyse der verschiedenen Sortieralgorithmen vermittelt typische *Analysetechniken*.

Definition des Sortierproblems

Es sei A ein Alphabet oder eine (nicht unbedingt endliche) geordnete Menge irgendwelcher Elemente.

Eine *Folge* $v = v_1 \dots v_n$ mit $v_i \in A$ heißt *geordnet* : $\Leftrightarrow v_i \leq v_{i+1}$ für $i = 1, \dots, n-1$, sie heißt *invers geordnet* : $\Leftrightarrow v_i \geq v_{i+1}$ für $i = 1, \dots, n-1$.

Sortierproblem: Gegeben sei eine Folge $v = v_1 v_2 \dots v_n \in A^n$.

Ordne v so um, daß eine geordnete (oder invers geordnete) Folge entsteht.

Genauere Formulierung: Gesucht wird eine *Permutation*

$\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$, so daß die Folge

$v_{\pi(1)} v_{\pi(2)} \dots v_{\pi(n)}$ geordnet ist.

Klassifizierung der grundlegenden Sortiermethoden

Es gibt mindestens die folgenden fünf grundlegenden *Sortiermethoden* (mit jeweils typischen, konkreten *Sortierverfahren*), nämlich "Sortieren durch ..."

(1) *Aussuchen*

Man sucht in der Folge $v_1 v_2 \dots v_n$ ein Element v_i mit geeigneten Eigenschaften (z.B. das kleinste oder größte Element). Man bringt dieses Element an die "richtige" Stelle und arbeitet mit der Restfolge nach der gleichen Idee rekursiv weiter.

Entsprechende konkrete Sortierverfahren sind z.B. Direktes Aussuchen, Quadratisches Aussuchen, Heapsort.

(2) *Einfügen*

Man setzt die Folge v Element für Element in eine Folge u um. Das jeweils nächste Element entfernt man aus v und fügt dieses in die bereits sortierte Anfangsfolge von u ein.

Entsprechende Sortierverfahren sind z.B. Direktes Einfügen, Binäres Einfügen.

(3) *Austauschen*

Man sucht in der Folge $v_1 \dots v_n$ zwei Elemente v_i und v_j mit $i < j$ und $v_i > v_j$ und vertauscht diese. Unterschiede ergeben sich in der Strategie des Auffindens solcher Elemente.

Entsprechende Sortierverfahren sind z.B. Bubblesort, Shakersort, Quicksort, Heapsort, Shellsort.

(4) *Mischen*

Wenn bereits zwei geordnete (oder teilweise geordnete) Folgen $v_1 \dots v_n \in A^n$ und $w_1 \dots w_m \in A^m$ vorliegen, dann kann man diese Folgen zu einer geordneten Folge der Länge $n+m$ "mischen", indem man die jeweils nächsten Elemente der beiden Folgen vergleicht und das kleinere der beiden Elemente übernimmt. Diese Idee ist vor allem für *externes Sortieren* sequentieller Dateien geeignet. Sortierverfahren dieser Art sind z.B. Zweiphasen-, Einphasen-, Natürliches Mischen.

(5) *Streuen und Sammeln*

Der Wertebereich, aus dem die zu sortierenden Elemente stammen, habe die Kardinalität m , d.h. $A = \{a_1, a_2, \dots, a_m\}$.

Wenn m nicht zu groß ist, dann kann man durch eine Funktion $ORD: A \rightarrow \text{INTEGER}$ die Folge $v = v_1 v_2 \dots v_n \in A^n$ auf ein $\text{ARRAY } [1..m] \text{ OF INTEGER}$ "streuen", indem man dieses Feld S mit 0 initialisiert und für $i = 1, 2, \dots, n$ die Wertzuweisung $S[ORD(v_i)] := S[ORD(v_i)] + 1$ durchführt, d.h. v_i in das durch die Funktion ORD bestimmte Feldelement ablegt. Anschließend

kann man in S mit $i = 1, \dots, m$ "sammeln", indem man für $S[i] \neq 0$ das Element a_i $S[i]$ -mal ausgibt.

Ein entsprechendes Sortierverfahren ist der Bucket-Sort.

Stabilität und Ordnungsverträglichkeit von Sortierverfahren

Eine erste, häufig gewünschte Eigenschaft von Sortierverfahren ist die der *Stabilität*:

Ein Sortierverfahren S heißt *stabil*, wenn S die Reihenfolge gleicher Elemente nicht verändert, d.h. wenn $S(v_1 \dots v_n) = v_{i_1} v_{i_2} \dots v_{i_n}$ ist, dann gilt für alle $v_{i_j} = v_{i_k}$ und $i_j < i_k$ stets $j < k$.

S heißt *invers stabil*, wenn die Reihenfolge der Elemente mit gleichem Schlüssel von S gespiegelt wird. Die Stabilität spielt eine Rolle, wenn man mehrstufig sortiert, z.B. erst nach Familien- und dann nach Vornamen. "Karl Maier" soll zum Schluß vor "Karl Müller" liegen.

Eine zweite, zentrale Eigenschaft von Sortierverfahren ist die *Ordnungsverträglichkeit*, für die wir zunächst einige Begriffe vorbereiten müssen.

Sei $S_n = \{\pi \mid \pi: \{1, \dots, n\} \rightarrow \{1, \dots, n\} \text{ ist eine Permutation}\}$ die Menge der *Permutationen* von n Elementen (in der Mathematik auch *symmetrische Gruppe* genannt). Es gilt: $|S_n| = n!$.

Eine Permutation π kann man als Wort

$$\pi(1) \pi(2) \dots \pi(n) \in \{1, \dots, n\}^*$$

eindeutig darstellen. Sei umgekehrt

$$\underline{S}_n = \{z_1 \dots z_n \mid z_i \in \{1, \dots, n\} \text{ und } z_i \neq z_j \text{ für } i \neq j\},$$

dann gehört zu jedem Wort $z \in \underline{S}_n$ genau eine Permutation $\pi \in S_n$ (nämlich $\pi(i) = z_i$ für $i = 1, \dots, n$) und umgekehrt: $|\underline{S}_n| = |S_n| = n!$.

Wir setzen deshalb π und z im folgenden gleich.

Für $\pi \in S_n$ sei

$$\alpha(\pi) = |\{(i, j) \text{ mit } i < j \text{ und } \pi(i) > \pi(j)\}|$$

die *Inversionszahl* von π (auch *Fehlstand* genannt).

Beispiele: $\alpha(1 \ 2 \ \dots \ n) = 0$

$\alpha(n \ n-1 \ \dots \ 2 \ 1) = \frac{1}{2} n(n-1)$, denn das Element n in der 1. Position steht bezüglich der folgenden $n-1$ Positionen "falsch", das Element $n-1$ in der 2. Position bezüglich der folgenden $n-2$ Positionen usw., d.h. wir haben

$$\text{die Summe } (n-1) + (n-2) + \dots + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{1}{2} n(n-1).$$

Nun können wir die Ordnungsverträglichkeit definieren:

Ein Sortierv Verfahren S heißt *ordnungsverträglich*: $\Leftrightarrow$

“je geordneter die zu sortierende Folge bereits ist, umso schneller arbeitet S ”, d.h. S berücksichtigt eine evtl. vorhandenen *Vorsortierung* der Folge.

Genauer: Wenn S für zwei Folgen $v_1 v_2 \dots v_n$ und $w_1 w_2 \dots w_n$ die Permutationen $\pi \in S_n$ und $\sigma \in S_n$ realisiert, und wenn die *Inversionszahl* $\alpha(\pi)$ kleiner als $\alpha(\sigma)$ ist, dann ist auch die *Schrittzahl*, die S zum Sortieren von $v_1 \dots v_n$ benötigt, kleiner (oder gleich) der Schrittzahl zum Sortieren von $w_1 \dots w_n$.

Formal: $S(v_1 \dots v_n) = v_{\pi(1)} v_{\pi(2)} \dots v_{\pi(n)}$ und
 $S(w_1 \dots w_n) = w_{\sigma(1)} w_{\sigma(2)} \dots w_{\sigma(n)}$
 und $\alpha(\pi) \leq \alpha(\sigma) \Rightarrow$ (Schrittzahl zum Sortieren von $v_1 \dots v_n$)
 $\leq$ (Schrittzahl zum Sortieren von $w_1 \dots w_n$)

Bemerkungen zum Sortieren “in der Praxis”

1. Die Elemente einer zu sortierenden Folge sind in der Praxis häufig umfangreiche *Records*. Die Folge wird jeweils nach einem *Ordnungskriterium* sortiert.

Fall 1: Hier bezieht sich das Ordnungskriterium *nicht* direkt auf eine Komponente des Records, sondern wird durch eine Funktion $g: A \rightarrow M$ beschrieben, wobei M eine geordnete Menge ist (Bsp. Personal-, Matrikel-, Versicherungsnummer). Es ist empfehlenswert, $g(v_i) := k_i$ durch Erweiterung in den Record-Typ aufzunehmen. k_i heißt dann (künstlicher) *Schlüssel* von (v_i) oder auch *Primärschlüssel* (jeden Satz eindeutig identifizierender Schlüssel).

Fall 2: Das Ordnungskriterium bezieht sich *direkt* auf eine oder mehrere Komponenten des Records. Den Vektor dieser Komponenten, nach denen zu sortieren ist, bezeichnen wir als (direkten) *Schlüssel*. Probleme: u.a. Eindeutigkeit, zu großer Aufwand für das Sortieren.

2. Wird eine Folge nach *mehreren* Record-Komponenten nacheinander geordnet, so kann man die Folge zunächst nach dem am wenigsten relevanten Kriterium und dann schrittweise nach dem nächstwichtigeren Kriterium sortieren. Hierfür muß ein Sortierv Verfahren allerdings *stabil* sein.
3. Sortieren erfordert in der Praxis viele Umspeicherungs-Operationen. Sind die Elemente sehr groß, so kostet dieses viel Zeit. Abhilfe verspricht hier folgende typische Technik: Man zieht die Schlüssel und den Verweis (Zeiger) auf das

jeweilige Datenelement aus dem Record heraus und sortiert nur diese *Schlüssel-Verweis-Tabelle*.

Zum Aufwand des Sortierens

Frage: Wie schnell kann man unter der Voraussetzung, als Basis-Operatoren nur den Vergleich und die Vertauschung jeweils zweier Elemente zur Verfügung zu haben, sortieren?

Eine Antwort liefert folgender Gedankengang:

Wenn die Sortierung aufgrund von *binären* Entscheidungen über Folgenglieder erfolgt, so wird durch diese *Abfragefolge* letztlich jede Permutation der Folge charakterisiert. Dann kann man die Permutationen als Blätter eines binären *Entscheidungsbaumes* auffassen, wobei in den inneren Knoten die Vergleiche der Folgenglieder stehen. Jede *Permutation* ist nun durch den *Weg* vom Blatt zur Wurzel (also: durch die Folge der auf diesen Weg stehenden Vergleiche) eindeutig bestimmt.

Diese Information kann man für den Fall einer zu sortierenden dreielementigen Folge $U_1 U_2 U_3$ den Blättern des Entscheidungsbaumes in Abb. 6.53 entnehmen.

Abb. 6.53 Entscheidungsbaum zur Bestimmung von Sortierfolgen

Für diesen Entscheidungsbaum gilt:

- Er besitzt $n!$ Blätter und $n! - 1$ innere Knoten.
 - Genau ein Knoten (nämlich die Wurzel) besitzt die Höhe 1.
 - Höchstens 2^{i-1} Knoten besitzen eine Höhe $\leq i$.
- $\Rightarrow$ Höchstens $n!$ Knoten besitzen eine Höhe $\leq \lceil \lg n! \rceil$.

Beim *vollständig ausgeglichenen Baum* mit $n!$ Blättern stehen alle Blätter mindestens auf der Stufe $\lg(n!) - 1$. Aus der *Stirlingschen Formel* (siehe Anhang A) folgt
 $\lg(n!) > k \cdot n \cdot \lg(n)$ mit positivem k .

Folglich hat auch die *mittlere* Stufe der Blätter diesen Wert. Ist der Baum *nicht* vollständig ausgeglichen (d.h. nicht alle Sortierungen sind gleichwahrscheinlich), so steigt die mittlere Stufe der Blätter. Der mittlere Aufwand zum Erreichen eines Blattes (d.h. einer Sortierung) kann also für Algorithmen, die ausschließlich auf binären Entscheidungen über Folgenglieder basieren, nicht kleiner sein als $O(n \lg n)$.

Mit diesem Gedankengang wird üblicherweise “bewiesen”, daß Sortieralgorithmen im Zeitverhalten nicht unter $O(n \lg n)$ gedrückt werden können (Randbedingungen: Betrachtung des durchschnittlichen Falls, d.h. alle Folgen sind gleichwahrscheinlich).

Der folgende Satz wird uns zeigen, daß man durch Vertauschen nur *direkt benachbarter* Elemente, d.h. Elemente dürfen nicht über größere Distanzen getauscht werden, sogar einen quadratischen Aufwand erhält.

Satz:

Jeder Sortieralgorithmus, der nur die Operatoren V_i (“Vertauschen direkt benachbarter Elemente der i -ten und $i+1$ -ten Stelle”) und Sp (“Spiegeln der gesamten Folge”) auf eine Folge $u = u_1 u_2 \dots u_n \in A^n$ anwenden darf, benötigt zur Sortierung von u im schlechtesten Fall mindestens $\left(\frac{1}{4} n (n - 1) - \frac{1}{2}\right)$ Schritte, also mit der Anzahl der zu sortierenden Elemente quadratisch wachsenden Aufwand (Bemerkung: das Spiegeln ist hier nur ein Zusatz-Operator, der das Sortieren zwar beschleunigt, aber den quadratischen Aufwand auch nicht verhindern kann).

Beweis:

Es sei $u_1 u_2 \dots u_n$ die zu sortierende Folge.

Dann gelten für die zugelassenen Operatoren

$Sp(u_1 \dots u_n) = u_n \dots u_1$ und

$V_i(u_1 \dots u_n) = u_1 \dots u_{i-1} u_{i+1} u_i u_{i+2} \dots u_n, i = 1, \dots, n-1$

die beiden folgenden Beobachtungen.

Beobachtung 1: $\alpha(u_1 \dots u_n) + \alpha(Sp(u_1 \dots u_n)) = \frac{1}{2} n (n-1),$

d.h. die Summe der Fehlstände einer Folge und ihrer “Spiegelung” ist *konstant* $\frac{1}{2} n (n-1)$, weil jedes u_i zu jedem anderen Element entweder im Original oder in der Spiegelung falsch steht.

Beobachtung 2: $\alpha(V_i(u_1 \dots u_n)) = \begin{cases} \alpha(u_1 \dots u_n) + 1, & \text{falls } u_i < u_{i+1} \\ \alpha(u_1 \dots u_n) - 1, & \text{falls } u_i > u_{i+1} \end{cases}$

d.h. eine Vertauschung direkt benachbarter Elemente erhöht den Fehlstand um 1 (falls die Elemente vorher in der richtigen Reihenfolge waren) oder erniedrigt den Fehlstand um 1 (falls die Elemente vorher in der falschen Reihenfolge waren).

Wegen Beobachtung 1 ist für eine Folge mit einem Fehlstand von $\frac{1}{4}n(n-1)$ der Aufwand für die Sortierung des Originals und der Spiegelung gleich groß (da sich dieser Fehlstand durch die Operation Sp nicht ändert). Dieser Wert kann als *Mittelwert* der beiden "Idealfälle" $\alpha(u_1 \dots u_n) = 0$ (*geordnete Folge*) und $\alpha(u_1 \dots u_n) = \frac{1}{2}n(n-1)$ (*invers geordnete Folge*) betrachtet werden und beschreibt deshalb die schlechteste Ausgangssituation. Als Gütekriterium einer teilweise sortierten Folge definieren wir deshalb den Betrag der Differenz zu diesem schlechtesten Wert des Fehlstandes einer Folge:

$$\beta(u_1 \dots u_n) := \left| \frac{1}{4}n(n-1) - \alpha(u_1 \dots u_n) \right|$$

Dann gilt wegen der Beobachtungen 1 und 2:

$$\begin{aligned} \beta(Sp(u_1 \dots u_n)) &= \beta(u_1 \dots u_n) \text{ und} \\ |\beta(V_i(u_1 \dots u_n)) - \beta(u_1 \dots u_n)| &= 1 \text{ und} \\ \beta(V_i(u_1 \dots u_n)) &\leq \beta(u_1 \dots u_n) + 1. \end{aligned}$$

Es seien $\eta_1, \eta_2, \dots, \eta_r \in \{V_1, V_2, \dots, V_{n-1}, Sp\}$ r Operatoren. Dann gilt:

$$\beta(\eta_1, \eta_2 \dots \eta_r(u_1 \dots u_n)) \leq \beta(u_1 \dots u_n) + r.$$

Es sei nun $u_1 \dots u_n$ eine Folge mit $\beta(u_1 \dots u_n) = 0$ (bzw. $\beta(u_1 \dots u_n) = \frac{1}{2}$, da $\frac{1}{4}n(n-1)$ nur für Vielfache von 4 einen ganzzahligen Wert hat). Dieser Wert entspricht einer maximal ungeordneten Folge, d.h. einer Folge mit maximaler Anzahl notwendiger Vertauschungen bzgl. der sortierten und der invers sortierten Folge.

$\eta_1 \dots \eta_r$ sei eine Folge von Operationen, die $u_1 \dots u_n$ in die geordnete Folge $u'_1 \dots u'_n$ mit $\beta(u'_1 \dots u'_n) = \frac{1}{4}n(n-1)$ überführt.

Dann gilt: $\frac{1}{4}n(n-1) = \beta(u'_1 \dots u'_n) = \beta(\eta_1 \dots \eta_r(u_1 \dots u_n)) \leq \beta(u_1 \dots u_n) + r.$

Also folgt $r \geq \frac{1}{4}n(n-1)$, bzw. $r \geq \frac{1}{4}n(n-1) - \frac{1}{2}.$

Ein Algorithmus benötigt aber mindestens r Schritte, woraus der Satz folgt.

Schnelle Sortiervverfahren müssen also die Elemente über *größere Entfernungen* (vergleichen und) vertauschen, um auch im schlechtesten Fall die Größenordnung $O(n \log n)$ zu erreichen. Später vorgestellte Sortiervverfahren wie z.B. Quicksort und Heapsort arbeiten nach diesem Prinzip und erreichen deshalb diese Größenordnung. Im Fall von Quicksort gilt diese Aussage - wie wir später zeigen werden - jedoch nur für den durchschnittlichen Fall.

Vergleich von Sortiervverfahren

Wenn Sortiervverfahren in Form konkreter Algorithmen verglichen werden, so muß man zunächst klären, welche Kriterien mit welcher Gewichtung einem solchen Vergleich zugrundegelegt werden. Dabei kommen z.B. in Frage:

- (a) die Anforderungen an die Speicherung der zu sortierenden Daten,
- (b) der Aufwand für Entwurf und Implementierung,
- (c) der Speicherbedarf für Code und Daten,
- (d) der Rechenzeitbedarf,
- (e) andere Merkmale wie die Stabilität und die Ordnungsverträglichkeit, also der Nutzen einer Vorsortierung.

Kriterium (a) ergibt eine Unterscheidung in zwei Kategorien, nämlich Sortiervverfahren für Daten, die im Hauptspeicher mit wahlfreiem Zugriff stehen (*interne Sortiervverfahren*) und solche, die auch mit Daten arbeiten können, die in sequentiellen Dateien stehen (*externe Sortiervverfahren*). Bei der internen Sortierung verlangt man in der Regel, daß die Sortierung *in situ*, also im vorhandenen Speicherplatz stattfindet; nur eine feste Zahl von Variablen darf zusätzlich verwendet werden. Bei der Verwendung externer Speicher kann man dagegen großzügig mit dem Speicherplatz (Kriterium (c)) umgehen. Der Speicherplatz für den Code spielt eine ganz untergeordnete Rolle und wird hier nicht weiter betrachtet. Bezüglich der Kriterien (c) und (d) kann dabei noch eine mehr oder minder starke Datenabhängigkeit bestehen.

Die Unterscheidung einfacher und schneller interner Sortiervverfahren betrifft sowohl Kriterium (b) als auch (d), d.h. die schnellen Verfahren sind schwieriger zu verstehen, die einfachen entsprechen den Methoden, die wir auch bei der "Sortierung von Hand" verwenden. Diese einfachen Verfahren sind solche, deren Aufwand von der Ordnung n^2 ist, während die schnellen - wie wir bereits diskutiert haben - den Aufwand $O(n \log n)$ erreichen.

Eine Vorsortierung (Punkt (e)) beschleunigt einige Verfahren erheblich, andere nicht. Wo zu sortierende Folgen nur einzelne "Fehlstellen" enthalten, ist dieser Aspekt von Bedeutung.

Die Effizienzvergleiche stützen sich vor allem auf die Anzahl der Vergleiche und Zuweisungen der zu sortierenden Elemente. Wir bezeichnen nachfolgend die Zahl der Vergleiche mit C (*Comparison*), die der Zuweisungen mit M (*Move*). Bei einigen

Sortierv Verfahren betrachten wir auch die Anzahl der Vertauschungen, die wir mit T (*Tauschoperationen*) bezeichnen. Eine Vertauschung kann jeweils durch drei Zuweisungen realisiert werden. Man beachte, daß die Zuweisungen und Vergleiche der Indizes in der Regel wesentlich effizienter sind und hier nicht berücksichtigt werden.

Der günstigste Fall (*best case*) wird durch das Suffix *min*, der ungünstigste (schlechteste) durch *max* (*worst case*), der durchschnittliche durch *ave* (*average case*) bezeichnet.

Wir werden vor allem die Verfahren zur internen Sortierung betrachten und die externe Sortierung nur kurz streifen. Angesichts der heute sehr großen Hauptspeicher ist der erste Fall auch wesentlich wichtiger.

6.5.2 Interne Sortierv Verfahren

Bemerkung: Zur Implementierung der vorgestellten Algorithmen siehe grundsätzlich 6.5.2.3.

6.5.2.1 Einfache Sortierv Verfahren

Direktes Aussuchen

Beim direkten Aussuchen (Straight Selection) oder auch direkten Auswählen wird der unsortierte Teil eines Feldes nach seinem *kleinsten* Element durchsucht, welches dann mit dem *ersten* Element des *unsortierten* Teils vertauscht wird. Die Länge des zu sortierenden Feldes verringert sich so bei jedem Durchlauf um 1. Abb. 6.54 veranschaulicht dieses Verfahren (der unsortierte Teil des Feldes ist grau unterlegt).

Aufwandsabschätzung:

Da das restliche Feld stets bis zum Ende durchsucht wird, ist der Aufwand nur von n , nicht vom Inhalt des Feldes abhängig.

$$C_{\min} = C_{\max} = C_{\text{ave}} = \sum_{i=1}^{n-1} (n-i) = n(n-1) - \frac{n}{2}(n-1) = \frac{n}{2}(n-1)$$

$$M_{\min} = M_{\max} = M_{\text{ave}} = \sum_{i=1}^{n-1} 3 = 3(n-1)$$

Ist (wie in den beiden letzten Zeilen von Abb. 6.54) bei einem Sortierschritt das nächste Element bereits am richtigen Platz, so müßte natürlich nicht vertauscht werden. Allerdings macht die zusätzlich Abfrage das Programm eher langsamer, so daß man diesen (i.a. relativ seltenen) Fall besser schematisch behandelt.

5	33	12	13	8	1	41
1	33	12	13	8	5	41
1	5	12	13	8	33	41
1	5	8	13	12	33	41
1	5	8	12	13	33	41
1	5	8	12	13	33	41

Abb. 6.54 Direktes Aussuchen

Direktes Einfügen

Beim direkten Einfügen (Straight Insertion) findet die Suche im sortierten Teil des Feldes statt. Das jeweils *erste* Element des unsortierten Teils wird im sortierten Teil *entsprechend seiner Größe* eingefügt, alle größeren bereits sortierten Elemente müssen verschoben werden. Die Suche nach der Position zum Einfügen wird mit der Verschiebung kombiniert, d.h. man durchläuft den sortierten Teil *rückwärts* und verschiebt, bis das neue Element eingefügt wird. Abb. 6.55 veranschaulicht dieses Verfahren.

Aufwandsabschätzung:

- a) *Bester Fall:* Feld bereits sortiert (nur überflüssige Vertauschungen am Platz)

$$M_{\min} = \sum_{i=2}^n 3 = 3(n-1)$$

$$C_{\min} = \sum_{i=2}^n 1 = n-1$$

- b) *Schlechtester Fall:* Feld absteigend sortiert

$$\begin{aligned} M_{\max} &= \sum_{i=2}^n (3+i-1) = 2(n-1) + \sum_{i=2}^n i = 2(n-1) + n \frac{n+1}{2} - 1 \\ &= \frac{1}{2}(n^2 + 5n - 6) \end{aligned}$$

$$C_{\max} = \sum_{i=2}^n i = \frac{n(n+1)}{2} - 1 = \frac{1}{2}(n^2 + n - 2)$$

c) *Durchschnittlicher Fall:*

Im Mittel fällt das Element i in die Mitte des sortierten Bereichs 1 bis $i-1$, verursacht also die Verschiebung von $\frac{i-1}{2}$ Einträgen. Bei jeder Suche wird ein Element mehr verglichen als verschoben.

$$M_{\text{ave}} = \sum_{i=2}^n \left(3 + \frac{i-1}{2}\right) = \frac{1}{4} (n^2 + 11n - 12)$$

$$C_{\text{ave}} = \sum_{i=2}^n \left(\frac{i-1}{2} + 1\right) = \frac{1}{4} (n^2 + 3n - 4)$$

5	33	12	13	8	1	41
5	33	12	13	8	1	41
5	12	33	13	8	1	41
5	12	13	33	8	1	41
5	8	12	13	33	1	41
1	5	8	12	13	33	41

Abb. 6.55 Direktes Einfügen

Binäres Einfügen

Binäres Einfügen funktioniert im Prinzip wie Direktes Einfügen, nur daß die Einfügestelle im sortierten Teilfeld *binär* gesucht wird.

Aufwandsabschätzung:

Die Anzahl der *Bewegungen* verändert sich im Vergleich zum Direkten Einfügen nicht. Die Zahl der notwendigen *Vergleiche* geht, da es sich um binäre Suche handelt, mit $\log m$, wenn m Elemente bereits sortiert sind. Für alle Elemente ergibt sich daher ein durchschnittlicher Aufwand von $C_{\text{ave}} \in O(n \log n)$.

Damit ist der Aufwand für Vergleiche geringer als beim Direkten Einfügen. Allerdings ist diese Verbesserung von geringem Wert, weil in der Regel die Wertzuweisungen dominieren. Diese benötigen weiterhin im Mittel quadratischen Aufwand. Binäres Einfügen bringt also bei komplizierterem Algorithmus kaum Gewinn. Bei guter Vorsortierung ist es sogar schlechter.

Bubblesort

Ein besonders einfacher Ansatz für das Sortieren besteht darin, jeweils *benachbarte* Elemente zu *vertauschen*, wenn sie nicht wie gewünscht geordnet sind. Dabei steigt das relativ größte Element wie eine "Blase" (bubble) im Wasser auf, was dem Verfahren seinen Namen gegeben hat.

Algorithmusentwurf:

```

REPEAT                                     (* Programm P6.17 *)
  Unveraendert := TRUE;
  FOR i := 2 TO Feldlaenge DO
    IF F[i-1] > F[i] THEN
      "Tausche Paar";  Unveraendert := FALSE;
    END (* IF *);
  END (* FOR *);
UNTIL Unveraendert;

```

Nach dem ersten Durchlauf steht das größte Element sicher am Ende. Daher besteht die Möglichkeit, den Indexbereich bei jedem Durchgang um 1 zu verkürzen. In diesem Fall lohnt es sich kaum, ein Flag Unveraendert zu überwachen; man iteriert, bis der Indexbereich die Länge 1 hat.

```

                                     (* Programm P6.18 *)
Obergrenze := Feldlaenge;
WHILE Obergrenze > 1 DO
  FOR i := 2 TO Obergrenze DO
    IF F[i-1] > F[i] THEN "Tausche Paar" END (* IF *);
  END (* FOR *);
  "Obergrenze dekrementieren";
END (* WHILE *);

```

Aufwandsabschätzung für diese Variante:

$$C_{\min} = C_{\max} = C_{\text{ave}} = \sum_{i=1}^{n-1} i = \frac{n}{2} (n-1)$$

$$M_{\min} = 0$$

$$M_{\max} = 3 \sum_{i=1}^{n-1} (n-i) = 3 \left(n(n-1) - \frac{n(n-1)}{2} \right) = \frac{3}{2} n (n-1)$$

$$M_{\text{ave}} = \frac{3}{4} n (n-1)$$

Wie man sieht, ist Bubblesort im *Durchschnitt* schlechter als Direktes Einfügen im schlechtesten Fall. Eine Variante von Bubblesort ist *Shakersort*: Hier wechseln die Blasen ihre Laufrichtung bei jedem Durchgang.

6.5.2.2 Schnelle Sortiervverfahren

Shellsort

Bei Shellsort wird in mehreren Stufen "von grob bis fein" (vor)sortiert. Dadurch braucht der feinste Sortiervorgang *weniger Vertauschungen* als ohne eine grobe *Vorsortierung*.

Anstelle des beim Sortieren durch Direktes Einfügen benutzten wiederholten Verschiebens um eine Position nach rechts bringt Shellsort die Elemente in größeren Sprüngen schneller an ihre endgültige Position. Dafür wählt man eine Folge von *Inkrementen* $h_t \geq h_{t-1} \geq \dots \geq h_2 \geq h_1 = 1$ ($t \geq 2$), d.h. abnehmenden Distanzen mit letzter Distanz 1, und sortiert nacheinander für $i = t, t-1, \dots, 2, 1$ die Folgen der Elemente im Abstand h_i voneinander (hier i -ter Durchlauf für den speziellen Fall, daß h_i das n ganzzahlig teilt):

V_1	V_{1+h_i}	$V_{1+2 h_i}$	...	$V_{1+n \cdot h_i}$
V_2	V_{2+h_i}	$V_{2+2 h_i}$	...	$V_{2+n \cdot h_i}$
.				
.				
.				
V_{h_i}	$V_{2 h_i}$	$V_{3 h_i}$	...	V_n

Beispiel: Inkremente 4, 2, 1 und $n = 8$.

44	55	12	42	94	18	06	67	Ausgangsfolge
44	18	06	42	94	55	12	67	Sortierung für $h_3=4$
06	18	12	42	44	55	94	67	Sortierung für $h_2=2$
06	12	18	42	44	55	67	94	Sortierung für $h_1=1$

Der folgende Algorithmus verwendet Direktes Einfügen als Sortieralgorithmus für die jeweiligen Folgen.

Algorithmus:

```

(* Programm P6.19 *)
FOR s := t TO 1 BY -1 DO      (* A von 1 bis n indiziert *)
  h := hs;                  (* h ist Inkrement-Feld *)
  Anzahl := n DIV h;          (* Anzahl Teilfolgeelemente *)
  FOR i := 1 TO h DO          (* Teilfolgen fuer Abstand h *)
    FOR k := 1 TO Anzahl-1 DO
      x := A[i + k*h];        (* einzusortierendes Element *)
      j := i + (k-1)*h;       (* Index des aktuellen
                              Vergleichselements *)

```

```

(* durch direktes Einfuegen x einsortieren. *)
WHILE (j > 0) AND (x <= A[j]) DO
  A[j + h] := A[j];
  j := j - h;
END (* WHILE *);
A[j + h] := x;
END (* FOR k *);
END (* FOR i *);
END (* FOR s *);

```

Dieser Algorithmus ist stabil, aber nicht ordnungsverträglich, seine Analyse recht kompliziert. Die Wahl der Distanzenfolge ist maßgebend für die Geschwindigkeit des Shellsorts. Als besonders günstig erwies sich die Folge $h_t = (h_{t+1} - 1)/2$. Die "beste Folge" wurde jedoch noch nicht gefunden.

Heapsort

Beim Direkten Aussuchen holen wir solange das *kleinste* Element aus einem sich verkleinernden Teilfeld, bis dieses leer und damit das Gesamtfeld sortiert ist. Die einzelnen Elemente sind also quasi an einer langen Schnur aufgereiht, die zuerst sehr verschlungen angeordnet ist und mit dem Sortieren langsam entwirrt wird. Dabei "sieht" man von jedem Element aus nur den Vorgänger und den Nachfolger. Mit mehr Informationen über die umgebenden Elemente ließe sich das Sortieren beschleunigen. Als eine verhältnismäßig einfache Möglichkeit bietet sich das (virtuelle) Anordnen der Elemente in einem *Binärbaum* an.

Ein Feld mit den Elementen 44, 55, 12, 42, 94, 18, 6, 67 kann zunächst wie in Abb. 6.56 dargestellt als Binärbaum interpretiert werden.

Abb. 6.56 Interpretation eines Feldes als Binärbaum (Indizes neben den Knoten)

Man sieht, daß jeder Knoten außer dem Knoten 1, der Wurzel, einen Vater hat, und zwar hat der Knoten j den Vater $(j \text{ DIV } 2)$.

Zum Sortieren muß daraus ein *Heap* gebildet werden, der durch folgende Eigenschaft gekennzeichnet ist:

$$\text{Element}_i \geq \text{MAX}(\text{Element}_{2i}, \text{Element}_{2i+1}) \quad (i = \text{Elementindex})$$

d.h. ein Knoten ist nie kleiner als seine Söhne, sofern er überhaupt Söhne hat.

Das zu sortierende Feld wird dieses Kriterium in der Regel zu Anfang nicht erfüllen, so daß wir, bevor wir mit der eigentlichen Sortierung beginnen können, zuerst einen Heap *aufbauen* müssen.

Die obere Hälfte des Feldes (Indizes $> \frac{n}{2}$) genügt, weil keine Söhne existieren, immer den Anforderungen, so daß wir mit dem Umgestalten erst von der Mitte abwärts (in Abb. 6.56 also für die Elemente 4 bis 1) beginnen müssen (vgl. auch Programm in 6.5.2.3).

Abb. 6.57 Aufbau des Heaps

Existieren für ein Element *beide* Söhne, so wird es gegebenenfalls gegen den größeren getauscht. Die 4. Vertauschung zieht rekursiv eine weitere nach sich (rechter Teil in Abb. 6.57). Danach ist das Neuordnen des Feldes abgeschlossen. Alle Elemente erfüllen jetzt das Heapkriterium (siehe Abb. 6.58).

Abb. 6.58 Vollständig geordneter Heap

In einem vollständigen Heap ist das erste Element immer auch das größte. Dieses tauschen wir nun gegen das letzte Feldelement aus (siehe Abb. 6.59, vertauscht werden die Elemente 1 und 8) und verkürzen den Heap um dieses Element (das damit richtig plziert ist). Das neu an die Spitze getauchte Element lassen wir nun in den Heap einsinken (Vertauschungen 1 und 1*), bis das Heapkriterium wieder erfüllt ist.

Abb. 6.59 Heapsort

Dieser Vorgang wird nun fortgesetzt, bis der Heap auf ein Element zusammengeschrumpft ist (und daher nicht mehr sortiert werden muß, letzte Zeile in Abb. 6.59). Danach ist das ganze Feld sortiert.

Aufwandsabschätzung:

Der Aufwand für Heapsort läßt sich nicht so leicht exakt berechnen wie bei den einfachen Sortierv Verfahren. Die folgende Überlegung zeigt einen relativ einfachen Ansatz für eine Abschätzung des ungünstigsten Falles, zunächst für die Vertauschungen, dann für die Vergleiche.

In der Aufbauphase kann der Knoten i höchstens so oft nach unten wandern, wie es Knoten darunter gibt. Beispielsweise erreicht der Inhalt des Knotens 1 in Abb. 6.56 beim Aufbau des Heaps (mit dem Schlüssel 44) durch zwei Vertauschungen seinen Zielplatz, im ungünstigsten Fall durch drei (wenn anstelle von 42 etwa 46 stände).

Beim Abbau wird jeder Knoten (bis auf den mit dem kleinsten Schlüssel) durch einen Tausch an seinen Zielplatz befördert. Dies ergibt insgesamt einen Beitrag zu $T_{\max}$ von $n-1$ (oder von $3(n-1)$ zu $M_{\max}$).

Ferner gelangt durch diesen Tausch aus jedem Knoten ein Element in die Wurzel und sinkt wieder in den Baum ein, und zwar höchstens bis zur Tiefe seines Vorgängers (wenn der Inhalt der Wurzel gegen Knoten i getauscht wird, ist dieser damit aus dem Baum herausgefallen, und die maximale Tiefe ist die von Knoten $i-1$). So tief kann der neue Inhalt der Wurzel maximal sinken. Nehmen wir zunächst vereinfachend an, daß die Tiefe der Knoten i und $i-1$ stets die gleiche sei, dann zieht also der Tausch bei Knoten i höchstens so viele Bewegungen nach sich, wie seiner Entfernung von der Wurzel entspricht.

Fassen wir nun die Überlegung zu Aufbau und Abbau zusammen, so sehen wir, daß der Knoten i insgesamt so viele Bewegungen erzeugt, wie seiner Distanz zu dem entferntesten Blatt und der zur Wurzel entspricht, also insgesamt der Pfadlänge von der Wurzel durch Knoten i zum Blatt. Der erwähnte Beitrag durch das Absondern kommt hinzu.

Nehmen wir an, daß genau $n = 2^m - 1$ Elemente zu sortieren sind und wir damit einen *vollständig ausgeglichenen Baum* haben. Dann stehen alle Blätter des Baums $m-1$ Stufen unter der Wurzel, und alle Pfadlängen sind $m-1$. Da es n Knoten gibt, folgt

$$T_{\max} = n(m-1) + (n-1).$$

Oben hatten wir angenommen, daß der Knoten $i-1$ stets die gleiche Höhe hat wie Knoten i . Tatsächlich ist bei den $m-1$ Knoten am linken Rand des Baums unter der Wurzel (mit den Nummern 2, 4, 8, ...) der Knoten mit der nächsttieferen Nummer um 1 weniger weit von der Wurzel entfernt. Dies ergibt eine Korrektur von $m-1$.

$$T_{\max} = n(m-1) + (n-1) - (m-1) = n \cdot m - m = (n-1) \cdot m$$

mit $m = \text{ld}(n+1)$ folgt daraus:

$$T_{\max} = (n-1) \text{ld}(n+1) \text{ und damit } M_{\max} = 3(n-1) \text{ld}(n+1).$$

Zur Abschätzung der Vergleiche: Die maximale Zahl der Vergleiche ist leicht abzuschätzen, weil jeder Tausch höchstens zwei Vergleiche erfordert (Sohn gegen Sohn, dann größerer Sohn gegen Vater). Die $n-1$ Vertauschungen beim Aushängen aus dem Heap erfordern keine Vergleiche. Folglich gilt:

$$C_{\max} = 2((n-1) \text{ld}(n+1) - (n-1)) = 2(n-1)(\text{ld}(n+1) - 1).$$

Tatsächlich ist dieser Wert etwas zu hoch, weil gelegentlich nur *ein* Vergleich notwendig ist.

Im günstigsten Fall ist die Tiefe beim Einsinken jeweils um 1 geringer, insgesamt sinkt damit die Zahl der Vertauschungen um $n-1$. Der gleiche Beitrag entsteht durch einen bereits geordneten Heap. Die Zahl der Vergleiche sinkt entsprechend, wobei der geordnete Heap gerade n Vergleiche beim Aufbau braucht (je 2 für $\frac{n}{2}$ Elemente).

Test: Für $n = 64$ wurde in 2.048 Versuchen mit Zufallszahlen als Durchschnitt ermittelt: $M_{\text{ave}} = 3 \cdot 329.8$, $C_{\text{ave}} = 578.8$ und 361.8 Aufrufe von Sift (vgl. 6.5.2.3).

Quicksort

Die hohe Leistung von Quicksort (Quicksort ist der schnellste bekannte interne Sortieralgorithmus) beruht auf der Tatsache, daß Austauschen am effizientesten ist, wenn es über große Distanzen erfolgt.

Man wählt aus dem Array irgendeinen Wert (im günstigsten Fall ist das der *Medianwert*) aus und läuft von der linken und der rechten Feldgrenze solange nach innen, bis ein nicht kleineres (auf der linken Seite) *und* ein nicht größeres (auf der rechten Seite) Element gefunden sind. Diese beiden werden dann vertauscht. Man wiederholt das Ganze solange, bis sich die beiden Indizes getroffen haben.

Jetzt hat man zwei Teilfelder, wobei das eine nur Elemente kleiner oder gleich dem Grenzwert und das andere nur Elemente größer oder gleich dem Grenzwert enthält. Diese beiden Teilfelder werden entsprechend dem oben beschriebenen Algorithmus *rekursiv* weiter sortiert, bis nur noch einelementige und damit naturgemäß sortierte Teilfelder vorhanden sind.

Quicksort wird durch den Aufruf der nachfolgenden rekursiven Prozedur Sort realisiert, wobei die Feldgrenzen als Parameter übergeben werden. Der Aufruf erfolgt nur, wenn das zu sortierende Feld mindestens zwei Elemente hat.

Algorithmus:

```

(* Programm P6.20 *)

PROCEDURE Sort (L, R: IndexTyp);
VAR left, right : IndexTyp;
    (* eigentlich [MIN (IndexTyp)-1..MAX (IndexTyp)+1]; *)
    Grenzwert : ElementTyp;
BEGIN (* Sort *)
    left := L;    right := R;
    Grenzwert := Sortierfeld [(L+R) DIV 2];    (* Index koennte
                                                auch Zufallszahl im Bereich L bis R sein *)
    REPEAT
        WHILE Sortierfeld [left] < Grenzwert DO INC (left); END;
        WHILE Sortierfeld [right] > Grenzwert DO DEC (right); END;
        IF left <= right THEN
            IF left < right THEN Tausch (left, right); END;
            INC (left);    DEC (right);    (* auch bei left = right *)
        END (* IF *);
    UNTIL left > right;

    IF L < right THEN Sort (L, right); END;
    IF R > left THEN Sort (left, R); END;
END Sort;

```

Die Arbeitsweise von Quicksort zeigt sich am besten bei grafischer Darstellung des Sortiervorgangs, wie ihn Abb. 6.60 präsentiert.

Abb. 6.60 Verlauf einer Sortierung mit Quicksort

Aufwandsabschätzung:

Im *ungünstigsten* Fall (worst case) ist die asymptotische Komplexität von Quicksort von der Ordnung n^2 . Dieser Fall entsteht, wenn bei jeder Partitionierung (oder doch bei einem nicht vernachlässigbaren Teil davon) gerade der maximale oder minimale im unsortierten Bereich vorkommende Wert als Grenzwert gewählt wird.

In diesem Fall findet pro Rekursionsstufe nur *ein* Tausch statt (das Grenzwert-Element wandert an den Rand). Das folgende Feld F zeigt ein Beispiel (oben die Indizes). Durch die Verwendung des mittleren Elementes ($F[(1+16) \text{ DIV } 2] = 1$) läuft right von 16 bis 8, left bleibt auf 1, und $F[1]$ und $F[8]$ tauschen den Platz (durch Fettdruck markiert). Sonst gibt es keine Bewegungen. Am Ende steht left auf 2, right auf 1. Das gleiche wiederholt sich mit $F[(2+16) \text{ DIV } 2] = 16$. zur rechten Seite hin, usw.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	5	6	7	8	1	16	9	10	11	12	13	14	15
1	3	4	5	6	7	8	2	16	9	10	11	12	13	14	15
1	3	4	5	6	7	8	2	15	9	10	11	12	13	14	16
1	2	4	5	6	7	8	3	15	9	10	11	12	13	14	16
1	2	4	5	6	7	8	3	14	9	10	11	12	13	15	16

und so fort bis

1	2	3	4	5	6		7	8		9	10	11	12	13	14	15	16
---	---	---	---	---	---	--	---	---	--	---	----	----	----	----	----	----	----

Nach einem weiteren Tausch ist dann die korrekte Ordnung erreicht.

Der Aufwand für Vergleiche ist in diesem Fall in jeder Rekursionsstufe (bis auf die beiden letzten) $R-L+3$, also beim ersten Male 18.

Dies ergibt mit der Korrektur für die Fälle $L-R \leq 2$: $C_{\max} = \frac{1}{2} \cdot (n+2)(n+3) - 9$. In diesem Fall sind aber nur $n-2$ Vertauschungen nötig.

Der "worst case" ist aber relativ unwahrscheinlich, im Mittel benötigt Quicksort nur einen Aufwand der Ordnung $n \lg(n)$. Beispielsweise hätte eine Teilung, die stets in der Mitte des Intervalls liegt, in jeder Rekursionsstufe r (von 0 bis $\lg(n) - 1$) im Mittel $\frac{n}{4}$ Vertauschungen zur Folge, denn bei der Hälfte der im halben Intervall liegenden

Werte ist ein Tausch nötig, also $T = \frac{n}{4} \cdot \lg(n)$. Da die Teilung aber jedes beliebige Verhältnis ergeben kann, entsteht ein anderer Faktor, den man wie folgt abschätzen kann:

Sei $T(n) = k \cdot n \cdot \lg(n)$. Liege ferner die Grenze zwischen den Partitionen rechts von $F[i]$. Über die Reihenfolge der Elemente des Sortierfeldes vor der Partitionierung existieren keine Aussagen. In diesem Fall geht man von der *Gleichverteilung* der Elemente aus, so daß die Wahrscheinlichkeit, daß ein bestimmtes Element an einer bestimmten Position steht, gleich $\frac{1}{n}$ ist. Die Wahrscheinlichkeit, daß ein bestimmtes Element an einer der Positionen 1 bis i steht, ist dann $q := \frac{i}{n}$. Dann gilt mit $t(n,i)$ als Zahl der notwendigen Tausch-Operationen:

$$t(n,i) = n \cdot q \cdot (1-q),$$

weil im Bereich 1 bis i (i Elemente) vor der Partitionierung nur im Mittel $q \cdot i$ Elemente waren, die in diesen Bereich gehören. Mit der Gleichung $i = q \cdot n$ folgt dann, daß $i - q \cdot i = n \cdot q \cdot (1-q)$ Elemente in diesen Bereich hineingetauscht worden sind. Aus der Rekursivität des Quicksort-Algorithmus (Anwendung von Quicksort auf die beiden neu entstehenden Teilintervalle) folgt dann für $T(n)$ die Gleichung:

$$T(n) = t(n,i) + T(i) + T(n-i).$$

Da i jeden Wert aus dem Bereich 1 bis $n-1$ mit gleicher Wahrscheinlichkeit annehmen kann, gilt der folgende Zusammenhang (*Mittelwertbildung*):

$$T_{\text{ave}}(n) = \frac{1}{n-1} \sum_{i=1}^{n-1} (t(n,i) + T_{\text{ave}}(i) + T_{\text{ave}}(n-i)).$$

Setzen wir nun für $T_{\text{ave}}(n)$ den Wert $k \cdot n \cdot \lg(n)$ ein, so folgt:

$$k \cdot n \cdot \lg(n) = \frac{1}{n-1} \sum_{i=1}^{n-1} n \cdot q \cdot (1-q) + \frac{1}{n-1} \sum_{i=1}^{n-1} (k \cdot q \cdot n \cdot (\lg(n) + \lg(q)) + k \cdot (1-q) \cdot n \cdot (\lg(n) + \lg(1-q))).$$

Durch Ausmultiplizieren und anschließendes Zusammenfassen der Summanden in der 2. Summe läßt sich diese Gleichung weiter vereinfachen. Da für $i=1, \dots, n$ die Terme q und $1-q$ den gleichen Wertebereich durchlaufen, liefern die beiden Komponenten in der zweiten Summe den gleichen Beitrag. Außerdem läßt sich als Teilsumme gerade die linke Seite der Gleichung herausziehen, so daß übrigbleibt:

$$0 = \frac{1}{n-1} \sum_{i=1}^{n-1} n \cdot q \cdot (1-q) + \frac{2}{n-1} \sum_{i=1}^{n-1} k \cdot q \cdot n \cdot \lg(q)$$

Diese Gleichung läßt sich nach k auflösen:

$$k = \frac{Z}{N} \text{ mit } Z = \sum_{i=1}^{n-1} q \cdot (1-q) \text{ und } N = -2 \sum_{i=1}^{n-1} q \cdot \lg(q)$$

Z kann leicht berechnet werden (unter Verwendung der Regel, daß die Summe der Quadrate bis n gleich $n \cdot (n+1) \cdot \frac{2n+1}{6}$ ist):

$$Z = \frac{1}{6} \cdot (n - \frac{1}{n}) \approx \frac{n}{6}$$

Einfacher ist die Annäherung durch das Integral, wobei q von 0 bis 1 läuft; sie ergibt genau $\frac{n}{6}$. Dabei muß jedoch berücksichtigt werden, daß aus $q = \frac{i}{n}$ die Substitution $di = n \cdot dq$ folgt. So gehen wir auch bei N vor und erhalten den Wert

$$N = -2 \cdot \frac{n}{\ln 2} \cdot (0 - \frac{1}{4}) = \frac{n}{2 \cdot \ln 2}, \text{ also } k = \frac{2 \cdot \ln 2}{6} = \frac{\ln 2}{3} \approx 0.231.$$

Die Zahl der Vergleiche pro Durchgang mit n Elementen ist gleich n (in 50% der Fälle um 1 höher). Die mittlere Zahl der Vertauschungen pro Durchgang ergibt sich aus der Mittelwertbildung über $t(n,i)$:

$$T_{\text{ave}}(n) = \frac{1}{n} \sum_{i=1}^n t(n,i) = \frac{1}{n} \sum_{i=1}^n n \cdot q \cdot (1-q) = \sum_{i=1}^{n-1} q \cdot (1-q) = Z = \frac{n}{6}$$

Damit ist $C_{\text{ave}} \approx 6 T_{\text{ave}}$, also

$$T_{\text{ave}} = \frac{\ln 2}{3} \cdot n \cdot \lg(n) = \frac{n}{3} \cdot \ln(n), \quad M_{\text{ave}} = 3 T_{\text{ave}}, \quad C_{\text{ave}} = 6 T_{\text{ave}}$$

Beispiel: Für $n = 64$ folgen daraus $T_{\text{ave}} = 88.7$ (89.9), $C_{\text{ave}} = 532$ (547).

In Klammern stehen die Resultate eines Tests mit 8.000 verschiedenen Zufallsbelegungen. Die Abweichungen entstehen vor allem durch den hohen Anteil der Sortierungen kürzester Partitionen, vor allem der Länge 2. Dabei gibt es im Mittel $\frac{1}{2} = \frac{n}{4}$

Vertauschungen (nicht $\frac{n}{6}$) und $\frac{5}{2}$ Vergleiche (also nicht $6T$). Die Zahl der Vertauschungen wird andererseits dadurch vermindert, daß gelegentlich der Grenzwert bereits am richtigen Ort steht.

6.5.2.3 Implementierung ausgewählter Sortiervverfahren

In diesem Abschnitt fassen wir in einem Modul Sortieren die Modula-2-Codierung aller vorgestellten Sortiervverfahren zusammen:

- Direktes Aussuchen
- Direktes Einfügen
- Binäres Einfügen
- Bubblesort
- Shellsort
- Heapsort
- Quicksort.

Allen entsprechenden Sortierprozeduren stehen die Prozedur Tausch zum *Vertauschen* zweier Elemente, eine Prozedur Generate zur Besetzung des Feldes mit *Zufallszahlen* und Hilfsprozeduren zur *Ausgabe* der Resultate zur Verfügung.

```

MODULE Sortieren;                                     (* Programm P6.21 *)
FROM InOut      IMPORT Read, WriteCard, WriteString, WriteLn;
FROM RandomM    IMPORT Random;                       (* kein Standard-Moduel *)
CONST MaxIndex  = 100;
TYPE IndexTyp   = [0..MaxIndex];                     (* 0 fuer Sentinel-Position *)
      ElementTyp = CARDINAL;                         (* oder anderer Typ mit
                                                    Ordnungsrelation *)
      FeldTyp    = ARRAY IndexTyp OF ElementTyp;
VAR  Sortierfeld, Kopie : FeldTyp;

PROCEDURE Tausch (i, j : IndexTyp); (* tauscht die Elemente i, j
                                     des Sortierfeldes *)
VAR Zwischenspeicher : ElementTyp;
BEGIN (* Tausch *)
  Zwischenspeicher := Sortierfeld [i];
  Sortierfeld [i] := Sortierfeld [j];
  Sortierfeld [j] := Zwischenspeicher;
END Tausch;

(***** DirektAussuchen *****)
PROCEDURE DirektAussuchen;
VAR i, j, k : IndexTyp;
BEGIN (* DirektAussuchen *)
  FOR i := 1 TO MaxIndex - 1 DO
    k := i;      (* kleinstes Element suchen, Index nach k *)
    FOR j := i + 1 TO MaxIndex DO
      IF Sortierfeld [j] < Sortierfeld [k] THEN k := j; END;
    END (* FOR *);
    Tausch (k, i);
  END (* FOR *);
END DirektAussuchen;

```

```

(***** DirektEinfuegen *****)
PROCEDURE DirektEinfuegen;
VAR i, j      : IndexTyp;
    Zwischenspeicher : ElementTyp;
BEGIN (* DirektEinfuegen *)
    FOR i := 2 TO MaxIndex DO
        Zwischenspeicher := Sortierfeld [i];
        Sortierfeld [0] := Zwischenspeicher;
        j := i;
        WHILE Zwischenspeicher < Sortierfeld [j-1] DO
            Sortierfeld [j] := Sortierfeld [j-1];    DEC (j);
        END (* WHILE *);
        Sortierfeld [j] := Zwischenspeicher;
    END (* FOR *);
END DirektEinfuegen;

(***** BinaerEinfuegen *****)
PROCEDURE BinaerEinfuegen;
VAR i, j, Mitte, L, R : IndexTyp;
    Zwischenspeicher : ElementTyp;
BEGIN (* BinaerEinfuegen *)
    FOR i := 2 TO MaxIndex DO
        Zwischenspeicher := Sortierfeld [i]; (* Einzufuegendes El. *)
        (* Einfuegestelle durch binaeres Suchen feststellen *)
        L := 1; (* Sortierfeld [i] wird nicht verglichen, *)
        R := i; (* denn Mitte bleibt stets kleiner als R. *)
        WHILE L < R DO
            Mitte := (L + R) DIV 2;
            IF Sortierfeld [Mitte] <= Zwischenspeicher THEN
                L := Mitte + 1;
            ELSE
                R := Mitte;
            END (* IF *);
        END (* WHILE *);
        (* hier zeigt R auf das erste Element, das groesser
           ist als Sortierfeld [i], oder R ist noch immer i *)
        FOR j := i TO R + 1 BY -1 DO (* Verschieben *)
            Sortierfeld [j] := Sortierfeld [j-1];
        END (* FOR *);
        Sortierfeld [R] := Zwischenspeicher; (* Einfuegen *)
    END (* FOR *);
END BinaerEinfuegen;

(***** Bubble-Sort *****)
PROCEDURE BubbleSort;
VAR i, j: IndexTyp;
BEGIN (* BubbleSort *)
    FOR i := 2 TO MaxIndex DO
        FOR j := MaxIndex TO i BY -1 DO
            IF Sortierfeld [j-1] > Sortierfeld [j] THEN
                Tausch (j-1, j);
            END (* IF *);
        END (* FOR *);
    END (* FOR *);
END BubbleSort;

```



```
(***** Shell-Sort *****)
```

```
PROCEDURE Shellsort;
CONST Passes          = 4;    (* Sortierlaeufe *)
TYPE PassTyp          = [1..Passes];
VAR i, j              : IndexTyp;
    Teilfolge         : IndexTyp;
    Anzahl            : IndexTyp;
    Delta             : IndexTyp;
    Deltas            : ARRAY PassTyp OF IndexTyp;
    Pass              : PassTyp;
    Zwischenspeicher  : ElementTyp;

BEGIN (* Shellsort *)
    Deltas [1] := 9;          Deltas [2] := 5;
    Deltas [3] := 3;          Deltas [4] := 1;
    FOR Pass := 1 TO Passes DO
        Delta := Deltas [Pass];
        Anzahl := MaxIndex DIV Delta;    (* Anzahl Teilfolgeele. *)
        FOR Teilfolge := 1 TO Delta DO    (* Teilfolgen f. Abstand *)
            FOR i := 1 TO Anzahl-1 DO
                Zwischenspeicher := Sortierfeld [Teilfolge + i * Delta];
                j := Teilfolge + (i-1) * Delta;    (* Index des aktuellen
                                                    Vergleichselements *)
                (* durch direktes Einfuegen x einsortieren. *)
                WHILE (j > 0) AND
                    (Zwischenspeicher <= Sortierfeld [j]) DO
                    Sortierfeld [j + Delta] := Sortierfeld [j];
                    j := j - Delta;
                END (* WHILE *);
                Sortierfeld [j + Delta] := Zwischenspeicher;
            END (* FOR i *);
        END (* FOR Teilfolge *);
    END (* FOR Pass *);
END Shellsort;
```

```
(***** Heap-Sort *****)
```

```
PROCEDURE HeapSort;    (* Elemente werden aufsteigend sortiert *)

PROCEDURE SiftRekursiv (Root, Max: IndexTyp);
VAR Sub : CARDINAL;
BEGIN (* SiftRekursiv *)
    Sub := 2 * Root;    (* Wurzel des einen Unterbaums *)
    IF Sub <= Max THEN    (* Unterbaum existiert *)
        IF (Sub < Max) AND
            (Sortierfeld [Sub+1] > Sortierfeld [Sub]) THEN
            INC (Sub);    (* groesseren Unterbaum auswahlen *)
        END (* IF *);
        IF Sortierfeld [Sub] > Sortierfeld [Root] THEN
            (* Wurzel faellt in den Unterbaum, falls kleiner *)
            Tausch (Sub, Root);
            SiftRekursiv (Sub, Max);    (* Wiederholung im Unterbaum *)
        END (* IF *);
    END (* IF *);
END SiftRekursiv;
```

```

VAR Index : IndexTyp;
BEGIN (* HeapSort *)
  FOR Index := MaxIndex DIV 2 TO 1 BY -1 DO    (* Heap aufbauen *)
    SiftRekursiv (Index, MaxIndex);
  END (* FOR *);
  FOR Index := MaxIndex - 1 TO 1 BY -1 DO      (* Heap abbauen *)
    Tausch (1, Index+1);
    SiftRekursiv (1, Index);                  (* verdraengtes Element
                                              in Heap setzen. *)
  END (* FOR *);
END HeapSort;

(***** Quick-Sort *****)
PROCEDURE QuickSort;

  PROCEDURE Sort (L, R: IndexTyp);            (* rekursive Sortierung *)
  VAR left, right : IndexTyp;
      (* eigentlich [MIN (IndexTyp)-1..MAX (IndexTyp)+1]; *)
  Grenzwert : ElementTyp;
  BEGIN (* Sort *)
    left := L;    right := R;
    Grenzwert := Sortierfeld [(L+R) DIV 2];  (* Index koennte auch
                                              Zufallszahl im Bereich L bis R sein *)
    REPEAT
      WHILE Sortierfeld [left] < Grenzwert DO INC (left); END;
      WHILE Sortierfeld [right] > Grenzwert DO DEC (right); END;
      IF left <= right THEN
        IF left < right THEN Tausch (left, right); END;
        INC (left); DEC (right);    (* auch bei left = right *)
      END (* IF *);
    UNTIL left > right;
    IF L < right THEN Sort (L, right); END;
    IF R > left THEN Sort (left, R); END;
  END Sort;

  (*
  (* sichere Variante von Quicksort ohne Rekursionsprobleme *)
  BEGIN (* Sort *)
    REPEAT
      left := L;    right := R;
      (* usw. wie oben bis: *)
    UNTIL left > right;
    (* kleineren Teil rekursiv,
      groesseren iterativ bearbeiten *)
    IF (right - L) < (R - left) THEN
      IF L < right THEN Sort (L, right); END;
      L := left;
    ELSE
      IF R > left THEN Sort (left, R); END;
      R := right;
    END (* IF *);
    UNTIL L >= R;
  END Sort;
  *)

```



```

BEGIN (* QuickSort *)          (* Start des rekursiven Sortierens *)
  IF MaxIndex > 1 THEN Sort (1, MaxIndex); END;
END QuickSort;

(***** Feld mit Zufallszahlen initialisieren *****)

PROCEDURE Generate (VAR Feld: FeldTyp);
VAR i : IndexTyp;
BEGIN (* Generate *)
  FOR i := 1 TO MaxIndex DO Feld [i] := Random (); END;
END Generate;

(***** Ausgabe der Resultate *****)

PROCEDURE Out (VAR Feld: FeldTyp);          (* Ausgabe des Feldes
auf Output *)
VAR i : CARDINAL;
BEGIN (* Out *)
  WriteLn; WriteLn;
  FOR i := 1 TO MaxIndex DO
    WriteCard (Feld [i], 6);
    IF (i MOD 10) = 0 THEN WriteLn; END;
    (* Max. 10 Zahlen pro Zeile *)
  END (* FOR *);
  WriteLn;
END Out;

TYPE SortProcType = PROCEDURE;
VAR RefFeld      : FeldTyp;
    Erster       : BOOLEAN;

PROCEDURE Check (Header   : ARRAY OF CHAR;
                 SortProc : SortProcType);

PROCEDURE Vergleich; (* Vergleich Soll <-> Ist-Ergebnisse *)
VAR i : CARDINAL; Fehler: BOOLEAN;
BEGIN (* Vergleich *)
  Fehler := FALSE;
  FOR i := 1 TO MaxIndex DO
    IF Sortierfeld [i] # RefFeld [i] THEN
      WriteString ('*** Fehler in Element ');
      WriteCard (i, 5); WriteLn;
      Fehler := TRUE;
    END (* IF *);
  END (* FOR *);
  IF NOT Fehler THEN
    WriteString
      ('Ergebnis stimmt mit erstem Ergebnis ueberein');
    WriteLn;
  END (* IF *);
END Vergleich;

```

```

BEGIN (* Check *)
  WriteLn;   WriteString (Header);
  Sortierfeld := Kopie;   SortProc;   Out (Sortierfeld);
  IF Erster THEN
    RefFeld := Sortierfeld;   Erster := FALSE;
  ELSE
    Vergleich;
  END (* IF *);
  Read (Zeichen);
END Check;

(***** Hauptprogramm *****)

BEGIN (* Sortieren *)
  Generate (Kopie);   Out (Kopie);   Erster := TRUE;
  Check ("Direkt Aussuchen", DirektAussuchen);
  Check ("Direkt Einfuegen", DirektEinfuegen);
  Check ("Binaer Einfuegen", BinaerEinfuegen);
  Check ("Bubble Sort", BubbleSort);
  Check ("ShellSort", ShellSort);
  Check ("HeapSort ", HeapSort);
  Check ("QuickSort", QuickSort);
END Sortieren.

```

6.5.2.4 Aufwandsvergleich der Sortiervverfahren

Zunächst ist anzumerken, daß nur die einfachen Sortieralgorithmen *stabil* sind (Ausnahme: Direktes Aussuchen), also die Reihenfolge mehrerer Elemente mit gleichem Schlüssel unverändert lassen. Eine bereits geordnete Eingabe führt bei allen Verfahren zu minimalem Aufwand, nur bei Heapsort ist dieser Fall eher ungünstig.

Tabelle 6.61 gibt einen Vergleich des Aufwands verschiedener Sortieralgorithmen. Die Angaben sind auf die Implementierung in 6.5.2.3 zugeschnitten; durch kleine Änderungen lassen sich noch Verbesserungen erzielen. Die Angaben zu den schnellen Algorithmen Heapsort und Quicksort sind nicht exakt. Außerdem fehlen bei Heapsort die Angaben für den durchschnittlichen Fall, da die entsprechenden Analysen sehr aufwendig sind. Da Heapsort jedoch noch im schlechtesten Fall einen Aufwand von $O(n \lg n)$ besitzt, sind die fehlenden Angaben nicht so interessant wie z.B. bei Quicksort. Man beachte, daß bei Quicksort $C_{\max}$ und $M_{\max}$ in unterschiedlichen Situationen auftreten.

Shellsort ist nur noch von akademischem Interesse und darum in der Tabelle nicht berücksichtigt. Binäres Einfügen unterscheidet sich von Direktem Einfügen nur durch einen ungefähr konstanten Aufwand für C .

Verfahren	$C_{\min}$	C_{ave}	$C_{\max}$	$M_{\min}$	M_{ave}	$M_{\max}$
Bubblesort	$\frac{n}{2}(n-1)$	$\frac{n}{2}(n-1)$	$\frac{n}{2}(n-1)$	0	$\frac{3n}{4}(n-1)$	$\frac{3n}{2}(n-1)$
Direktes Einfügen	$n-1$	$\frac{n^2+3n-4}{4}$	$\frac{n^2+n-2}{2}$	$3(n-1)$	$\frac{n^2+11n-12}{4}$	$\frac{n^2+5n-6}{2}$
Direktes Aussuchen	$\frac{n}{2}(n-1)$	$\frac{n}{2}(n-1)$	$\frac{n}{2}(n-1)$	$3(n-1)$	$3(n-1)$	$3(n-1)$
Heapsort	$C_{\max} - \frac{5}{2}n$		$2(n-1) \cdot (\lg(n+1)-1)$	$M_{\max} - 6n$		$3(n-1) \cdot \lg(n+1)$
Quicksort	$n \lg(n)$	$2n \cdot \ln(n)$	$\frac{1}{2}(n+3) \cdot (n+2) - 9$	0	$n \cdot \ln(n)$	$\frac{3}{4}n \lg(n)$

Tab. 6.61 Aufwand wichtiger Sortieralgorithmen

6.5.2.5 Sortieren durch Streuen und Sammeln

Wir haben festgestellt, daß Sortieren $O(n \log n)$ Schritte erfordert, wenn es auf binären Vergleichen beruht (und keine Informationen über die zu sortierenden Schlüssel vorliegen).

Beim Sortieren durch Streuen und Sammeln (mit *Bucketsort* als Vertreter) erfolgt eine Beschleunigung, weil der Wertebereich der Schlüssel als Intervall gegeben ist. Für die Sortierung wird dieses Intervall in endlich viele Teilintervalle ("Fächer", "Buckets") aufgeteilt. Das Sortieren durch Streuen und Sammeln ist durch eine *Streuphase* (Verteilen der Elemente auf die Fächer) und eine *Sammelphase* (Zusammenfügen der Elemente in den einzelnen Fächern zu einer neuen Gesamtfolge) charakterisiert.

Genauer: Es seien n verschiedene Elemente $a_1, a_2, \dots, a_n$ mit $\min \leq a_i \leq \max$ für $i \in \{1, \dots, n\}$ zu sortieren. Wir wollen ohne Beschränkung der Allgemeinheit annehmen, daß es sich um natürliche Zahlen handelt und $\min = 0$ bzw. $\max = m-1$ ist.

Die folgenden Algorithmen sind nur in Pseudocode beschrieben, da die exakte Implementierung der Buckets selbst nicht angegeben wird. Ein Bucket wird hier als *String* über einem festgelegten Basistyp mit den folgenden Konstanten und Operationen verwendet:

STRING OF [Typ]	≡	Definition eines Buckets mit Basistyp Typ,
ε	≡	leerer Bucket (mit beliebigem Basistyp),
o	≡	Anhängen eines neuen Elementes,
first	≡	Zugriff auf das erste Element,
rest	≡	Entfernen des ersten Elements.

Algorithmusentwurf (Bucketsort):

```

PROCEDURE BucketSort1;                                (* Programm P6.22 *)
CONST  n = 100;                                       (* Anzahl Elemente, also Feldlänge *)
      m = 100;                                       (* Dimension des Wertbereichs *)
VAR    Feld   : ARRAY [1..n] OF [0..m-1];
      Bucket : ARRAY [0..m-1] OF STRING OF [0 .. m-1];
      i      : CARDINAL;

BEGIN (* BucketSort1 *)
  FOR i := 0 TO m-1 DO                                (* Initialisierung *)
    Bucket [i] := ε;
  END (* FOR *);
  FOR i := 1 TO n DO                                  (* Streuen *)
    Bucket [A[i]] := Bucket [Feld[i]] o Feld[i]);
  END (* FOR *);
  j := 0;
  FOR i := 0 TO m-1 DO                                (* Sammeln *)
    WHILE Bucket [i] # ε DO
      INC(j);
      Feld[j] := first (Bucket [i]);
      Bucket [i] := rest (Bucket [i]);
      (* evtl. Mehrfach-Auftreten eines A[i] *)
    END (* WHILE *);
  END (* FOR *);
END BucketSort1;

```

Zeitkomplexität: $O(n+m)$.

Ein Problem dieses Sortierverfahrens ist die Größe von m , d.h. die Anzahl der Fächer, da mit ihr auch der Speicherplatzbedarf wächst. Ist die Anzahl verschiedener Werte im Feld größer als die Anzahl der Fächer (Bucket-Dimension), dann treten sogenannte *Kollisionen* auf, d.h. mehrere verschiedene $Feld[i]$ werden in *ein* Fach eingefügt.

Den Algorithmus kann man verallgemeinern und zur *Sortierung von k -Tupeln* für festes k verwenden, z.B. zur Sortierung k -stelliger Zahlen, wobei als Ordnung die übliche *lexikographische Ordnung* zugrunde gelegt wird. Dabei sortiert man zuerst nach der letzten Stelle, dann nach der vorletzten Stelle usw. bis zur ersten Stelle.

Lexikographische Ordnung:

Sei $A = \{a_1, a_2, \dots, a_n\}$ ein Alphabet, $v, w \in A^*$.

Auf A sei eine Ordnung, etwa $a_1 < a_2 < \dots < a_n$ vorgegeben, die sich wie folgt auf A^* fortsetzt:

$$v \leq w : \Leftrightarrow \begin{aligned} & (w = v u \text{ mit } u \in A^*) \text{ oder} \\ & (v = u a_i u' \text{ und } w = u a_j u'' \text{ mit } u, u', u'' \in A^* \\ & \text{und } a_i, a_j \in A \text{ mit } i < j) \end{aligned}$$

Die antisymmetrische Relation " $\leq$ " heit *lexikographische Ordnung*.

Im folgenden Programm P6.23 sei S ein String von n k -Tupeln ber dem Alphabet $\{0, \dots, m-1\}$.

Algorithmusentwurf (Bucketsort fr Tupel der Lnge k):

```

PROCEDURE BucketSort2;                                (* Programm P6.23 *)
(* Bucket sei ein Feld von Buckets fuer Tupel der Laenge k *)
VAR i, j : CARDINAL;

BEGIN (* BucketSort2 *)
  FOR i := k TO 1 BY -1 DO
    FOR j := 0 TO m-1 DO Bucket [j] := ε; END (* FOR *);
    WHILE S # ε DO                                     (* Streuen *)
      x := first (S); S := rest(S);
      (* Sei x = ( aj1, ..., ajk ) *)
      Bucket [aj1] := Bucket [aj1] ◦ x;
    END (* WHILE *);
    FOR j := 0 TO m-1 DO (* Sammeln *)
      S := S ◦ Bucket [aj];
    END (* FOR *);
  END (* FOR *);
END BucketSort2;

```

Zeitkomplexitt : $O(k(m+n))$. Fr festes k ist der Aufwand also $O(m+n)$.

Diese Idee lt sich auch auf Tupelfolgen mit Tupeln *unterschiedlicher Lnge* anwenden. Vor dem eigentlichen Sortieren werden die Tupel zunchst der Lnge nach *absteigend geordnet*. Dann werden nach obigem Verfahren die Tupel mit maximaler Lnge nach der letzten Stelle geordnet. Diese geordnete Liste wird an die Liste der Tupel gehngt, deren Lnge um 1 kleiner ist. Diese gesamte Liste wird nach der letzten Stelle der krzeren Tupel sortiert usw.

Beispiel:

- *Ausgangsfolge:* $S = (1,0,2) (1,0,2,1,0) (2,1) (0) (1,1,1) (2,0,1) (1,1,0) (1,0)$.
- Tupel der Länge nach absteigend sortieren:

LÄNGE [5] = (1,0,2,1,0)

LÄNGE [4] = ϵ

LÄNGE [3] = (1,0,2) (1,1,1) (2,0,1) (1,1,0)

LÄNGE [2] = (2,1) (1,0)

LÄNGE [1] = (0)

- Sortierung der Tupel nach Werten (von hinten nach vorne)

Durchlauf für 5. Stelle

$S = (1,0,2,1,0)$

Bucket [0] = (1,0,2,1,0); Bucket [1] = ϵ ; Bucket [2] = ϵ

Durchlauf für 4. Stelle

$S = (1,0,2,1,0)$

Bucket [0] = ϵ ; Bucket [1] = (1,0,2,1,0); Bucket [2] = ϵ

Durchlauf für 3. Stelle

$S = (1,0,2) (1,1,1) (2,0,1) (1,1,0) (1,0,2,1,0)$

Bucket [0] = (1,1,0); Bucket [1] = (1,1,1) (2,0,1)

Bucket [2] = (1,0,2) (1,0,2,1,0)

Durchlauf für 2. Stelle

$S = (2,1) (1,0) (1,1,0) (1,1,1) (2,0,1) (1,0,2) (1,0,2,1,0)$

Bucket [0] = (1,0) (2,0,1) (1,0,2) (1,0,2,1,0)

Bucket [1] = (2,1) (1,1,0) (1,1,1); Bucket [2] = ϵ

Durchlauf für 1. Stelle

$S = (0) (1,0) (2,0,1) (1,0,2) (1,0,2,1,0) (2,1) (1,1,0) (1,1,1)$

Bucket [0] = (0)

Bucket [1] = (1,0) (1,0,2) (1,0,2,1,0) (1,1,0) (1,1,1)

Bucket [2] = (2,0,1) (2,1)

- *Ergebnis:* $S = (0) (1,0) (1,0,2) (1,0,2,1,0) (1,1,0) (1,1,1) (2,0,1) (2,1)$.

Algorithmusentwurf (Bucketsort für Tupel unterschiedlicher Länge):

```

PROCEDURE BucketSort3;                                (* Programm P6.24 *)
(* LAENGE sei ein Feld von Buckets für variabel lange Tupel;
   in LAENGE[i] werden alle Tupel der Laenge i gespeichert. *)
VAR i, j, k : CARDINAL;
BEGIN (* BucketSort3 *)
  k := 0;                                              (* maximale Tupellaenge *)
  WHILE S #  $\epsilon$  DO
    x := first (S);  S := rest (S);
    p := |x|;      k := max {k,p};

```



```

    LAENGE [p] := LAENGE [p] o x;
  END (* WHILE *);
  FOR i := k TO 1 BY -1 DO
    S := LAENGE [i] o S;
    WHILE S # ε DO                                (* Streuen *)
      x := first (S);    S := rest (S);
                        (* Sei x = aj1, ..., aji, ..., ajp *)
      Bucket [aji] := Bucket [aji] o x;
    END (* WHILE *);
    FOR j := 0 TO m-1 DO                            (* Sammeln *)
      S := S o Bucket [j];
      Bucket [j] := ε;
    END (* FOR *);
  END (* FOR *);
END BucketSort3;

```

Zeitkomplexität: Wenn q die Gesamtlänge von S (d.h. die Summe der Längen aller Tupel) ist, dann beträgt der Zeitaufwand des Algorithmus $O(q+m)$.

6.5.3 Externe Sortierverfahren

Wir gehen nun davon aus, daß die zu sortierenden Folgen so groß sind, daß sie auf externen, nur sequentiell zugreifbaren Speichermedien wie Platten oder Bändern vorliegen und auch darauf zu sortieren sind.

Ein zentrales Anliegen dieser externen Sortierverfahren ist es, die Zahl der *Dateien* (Files) mit den zu sortierenden Folgen, vor allem aber die Zahl der Lese- und Schreiboperationen zu minimieren. Als Beispiele externer Sortierverfahren stellen wir das Direkte, das Natürliche und das Mehrwege-Mischen vor.

Zwei bereits sortierte Folgen/Sequenzen (*Läufe*) lassen sich zu einer verbinden (*"mischen"*), indem man jeweils das kleinere der in den *Ausgangsläufen* anstehenden Elemente in den *Ergebnislauf* übernimmt. Dieses sogenannte Mischen ist besonders geeignet, wenn man jeder Sequenz eine Datei zuordnet. Ein Beispiel für das Mischen zweier Ausgangsläufe 1 und 2 zu einem Ergebnislauf:

```

Lauf 1:   2 12 13 15 20 29
           ⇒ 2 4 5 9 12 13 14 15 20 22 29 (Ergebnislauf)
Lauf 2:   4 5 9 14 22

```

Die Technik des Mischens muß nun aber erweitert werden, um folgendes allgemeine externe Sortierproblem zu lösen: Gegeben sei ein *File* unbekannter Länge, dessen Elemente sortiert werden sollen.

6.5.3.1 Direktes Mischen

Das *Direkte Mischen* ist durch einen ständigen Wechsel zwischen der *Trennphase* (*Verteilung* der Läufe auf meist zwei sequentielle Dateien) und der anschließenden *Mischphase* gekennzeichnet und wird darum auch als "Zweiphasen-Mischen" bezeichnet. Die beiden Phasen bilden zusammen einen *Durchlauf*. Charakteristisch für das Direkte Mischen ist, daß sich durch jeden Durchlauf die Länge der Ausgangsläufe verdoppelt, beginnend bei 1 für eine unsortierte Folge. Daraus folgt, daß nach maximal $\lg(n) + 1$ Durchläufen die gesamte Sequenz sortiert ist.

Nachfolgend sind die Läufe durch "I" voneinander abgegrenzt.

File S:	7 I 13 I 9 I 4 I 25 I 1 I 30 I 12
<i>Trennen</i>	
Hilfsfile 1:	7 I 9 I 25 I 30
Hilfsfile 2:	13 I 4 I 1 I 12
<i>Mischen</i>	
File S:	7 13 I 4 9 I 1 25 I 12 30
<i>Trennen</i>	
Hilfsfile 1:	7 13 I 1 25
Hilfsfile 2:	4 9 I 12 30
<i>Mischen</i>	
File S:	4 7 9 13 I 1 12 25 30
<i>Trennen</i>	
Hilfsfile 1:	4 7 9 13
Hilfsfile 2:	1 12 25 30
<i>Mischen</i>	
File S:	1 4 7 9 12 13 25 30

Algorithmus:

```

(* Programm P6.25 *)

PROCEDURE Direkt2PhasenMischen
  (VAR Liste : ARRAY OF CARDINAL);
VAR  LaengeRun  : INTEGER;      (* Laenge der Laeufe *)
      NoOfRuns   : INTEGER;      (* Zahl der Laeufe *)
      Hilfslisten : ARRAY [1..2], [0..MaxLng] OF CARDINAL;
BEGIN (* Direkt2PhasenMischen *)
  LaengeRun := 1;
  REPEAT
    Trennen (LaengeRun);
    Mischen (LaengeRun, NoOfRuns (* Ergebnisparameter *));
    LaengeRun := 2 * LaengeRun;
  UNTIL NoOfRuns = 1;
END Direkt2PhasenMischen;
```


6.5.3.2 Natürliches Mischen

Beim Direkten Mischen wird eine zufällig oder durch frühere Bearbeitung bereits vorhandene Sortierung der Dateien nicht ausgenutzt. Die Länge aller gemischten Teilsequenzen nach dem k -ten Durchlauf ist gleich 2^k . Zwei beliebig, geordnete Teilsequenzen der Längen m und n könnten jedoch direkt zu einer geordneten Sequenz der Länge $n+m$ gemischt werden. Beim *Natürlichen Mischen* werden nicht Sequenzen fester Länge, sondern die jeweils vorgefundenen Teilsequenzen *maximaler* Länge gemischt.

File S:	7	13	9	4	25	1	30	12
<i>Trennen</i>								
Hilfsfile 1:	7	13	4	25	12			
Hilfsfile 2:	9	1	30					
<i>Mischen</i>								
File S:	7	9	13	1	4	25	30	12
<i>Trennen</i>								
Hilfsfile 1:	7	9	13	12				
Hilfsfile 2:	1	4	25	30				
<i>Mischen</i>								
File S:	1	4	7	9	13	25	30	12
<i>Trennen</i>								
Hilfsfile 1:	1	4	7	9	13	25	30	
Hilfsfile 2:	12							
<i>Mischen</i>								
File S:	1	4	7	9	12	13	25	30

Im Vergleich zum Direkten Mischen ($\lg(n) + 1$ Durchläufe) erfordert Natürliches Mischen einen wesentlich aufwendigeren Algorithmus, bringt aber außer bei sehr langen natürlichen Sequenzen kaum eine Leistungssteigerung, denn im unsortierten Feld haben die natürlichen Sequenzen im Mittel die Länge 2, so daß nur ein einziger Durchlauf einspart wird.

6.5.3.3 Mehrwege-Mischen

Das Prinzip des Mischens läßt sich effizienter implementieren, indem man jeweils auf das Trennen der Läufe verzichtet (bis auf den ersten, der unvermeidlich ist) und beim Mischen sofort wieder trennt, also z.B. zwei Files mischt und die entstehenden Läufe *abwechselnd* in zwei andere Files schreibt. Dann wiederholt sich der Vorgang in der *umgekehrten* Richtung, bis nur noch eine einzige Sequenz entsteht.

Dieses Prinzip läßt sich bei drei Dateien wie folgt anwenden: Man verteilt die Läufe so, daß zu jeder Zeit von zwei Dateien auf die dritte gemischt wird.

File S: 7 | 13 | 9 | 4 | 25 | 1 | 30 | 12

Trennen zur Initialisierung

Hilfsfile 1: 7 | 9 | 25

Hilfsfile 2: 13 | 4 | 1 | 30 | 12

Mischen der ersten drei Paare von Hilfsfiles 1 und 2 nach 3

Hilfsfile 3: 7 13 | 4 9 | 1 25

Mischen der ersten zwei Paare von Hilfsfiles 2 und 3 nach 1

Hilfsfile 1: 7 13 30 | 4 9 12

Mischen des ersten Paares von Hilfsfiles 1 und 3 nach 2

Hilfsfile 2: 1 7 13 25 30

Mischen des ersten Paares von Hilfsfiles 1 und 2 nach File S

File S: 1 4 7 9 12 13 25 30

Wie man sieht, ist die Zahl der Läufe auf den beiden nichtleeren Dateien jeweils ein Paar aufeinanderfolgender *Fibonacci-Zahlen* (hier 5 und 3, dann 3 und 2, dann 2 und 1, dann 1 und 1, dann 1 und 0, d.h. es gibt nur noch einen sortierten Lauf). Natürlich kann in beiden Algorithmen File S auch als Hilfsfile 3 verwendet werden.

Die Verteilung der Läufe als Paare aufeinanderfolgender Fibonacci-Zahlen ist kein Zufall. Durch eine solche Verteilung wird die Anzahl der Mischvorgänge für eine gegebene Anzahl von Ausgangsläufen *minimiert*. Aber genau hier liegen große Probleme, da i.a. unbekannt ist, wie groß das zu sortierende File ist, man also selten eine Fibonacci-Verteilung erhält und dann "Restelemente" verteilen muß, was erhebliche Schwierigkeiten bereiten kann.

6.6 Speicherverwaltung

Die Speicherverwaltung ist eine Komponente des *Betriebssystems*, deren Aufgabe die Zuweisung und Überwachung aller belegten (und damit implizit auch der freien) Speicherbereiche ist. Dafür werden typischerweise Tabellen angelegt, in denen die belegten und freien Speicherbereiche verwaltet werden.

Da i.a. nicht ein ganzes Programm und alle benötigten Daten in den *Hauptspeicher* passen, werden Teile auf einen *Hintergrundspeicher* (virtueller Speicher) ausgelagert. Zur Speicherverwaltung gehört deshalb sowohl die Hauptspeicher- als auch die Hintergrundspeicherverwaltung.

Ausgangssituation: Der Hauptspeicher sei in *Blöcke* (kleine Speicherbereiche fester Länge) zerlegt, die z.B. als lineare *Liste* verkettet sind. Die Blöcke können zeitweise "frei" oder "belegt" sein.

Eine typische Datenstruktur für Blöcke sieht wie folgt aus:

```

TYPE   BPtr  =  POINTER TO Block;
       Block =  RECORD
           groesse : [0 .. maxgroesse];
                (* Komponenten, die "groesse"
                  Speicherplaetze einnehmen. *)
           next : BPtr;
       END;

```

Diese "Block"-Struktur ist *dynamisch*, da ihre Größe von *groesse* abhängt (in Sprachen wie MODULA-2 nicht erlaubt). Zusätzlich benötigen wir die folgenden Variablen, die auf den jeweils ersten Block einer Liste mit den freien (Freiliste) bzw. belegten (Belegliste) Blöcken verweisen.

```
VAR Freiliste, Belegliste : BPtr;
```

Typische *Aufgaben* einer Speicherverwaltung sind

- *Speicherplatzanforderung*: Zu einer Anforderung nach einem Block der Größe k soll ein freier Block der Größe $\geq k$ gefunden werden.
- *Blockverschmelzung*: Benachbarte Blöcke, die als "frei" in die Freiliste eingefügt wurden, sollen zu einem Block verschmolzen werden.
- *Seitenaustauschverfahren*: Blöcke, die freigegeben werden können, sollen gegen aktuell interessante Daten, die auf dem Hintergrundspeicher liegen, ausgetauscht werden (engl. paging).

- **Speicherbereinigung:** Nicht mehr belegte Blöcke (genauer: Blöcke aktuell nicht interessierender Daten) sollen erkannt und als "frei" gekennzeichnet, d.h. der Freiliste zur Verfügung gestellt werden (engl. garbage collection)..

Die grundlegenden algorithmischen Konzepte zur Lösung dieser vier Aufgaben werden im folgenden vorgestellt.

6.6.1 Algorithmische Konzepte

Speicherplatzanforderung

Wir gehen von der in Abb. 6.62 dargestellten Situation aus: es gibt vier freie (1, 2, 4, 7) und vier belegte (3, 5, 6, 8) Blöcke, die aktuelle Größe des Blockes i ist durch g_i bestimmt. Es wird ein Block der Größe k oder größer gesucht.

Abb. 6.62 Beispiel eines Speichers mit freien und belegten Blöcken

Algorithmus first-fit

Man geht ab Freiliste[^] die Liste der freien Blöcke durch und weist den *ersten* Block, dessen Größe größer als k ist, zu.

Algorithmus best-fit

Man sucht - in der gleichen Liste - den Block mit der *kleinsten* Größe, die größer oder gleich k ist, und weist ihn zu.

Algorithmus mit Blockspaltung

Es wird zunächst ein Block der Größe g (nach Methode first-fit oder best-fit) bestimmt. Der Block wird dann *gespalten* in einen Block der Größe k und einen Block der Größe $(g-k-h)$, wobei h der für die *Blockverwaltung* benötigte Speicherplatz ist.

Nachteil:

Der Speicher wird nach einiger Zeit zu stark durch viele kleine Blöcke belegt. Insbesondere die Kombination mit best-fit erweist sich in der Praxis als ungeeignet.

Verbesserung: first-fit als Grundmethode sollte man so modifizieren, daß man nicht immer von vorne durchsucht, sondern ab der Stelle, wo zuletzt ein Block entnommen wurde (d.h. die Liste freier Blöcke sollte man *zyklisch* organisieren). Die so verbesserte Methode heißt *rotating-first-fit*.

Blockverschmelzung

Für das Wiedereinfügen von freien Blöcken gibt es folgende grundlegenden Möglichkeiten:

1. Am *Ende* (oder am *Anfang*) der Liste der freien Blöcke anhängen.
2. Feststellen, ob die beiden im Speicher physisch *benachbarten* Blöcke frei sind, und mit einem oder mit beiden verschmelzen; andernfalls wie unter 1. verfahren.
3. Um das Suchen nach einem geeigneten Block zu verkürzen, kann man die Blöcke der Größe nach *sortiert* verketteten. Das Wiedereinfügen entspricht dann dem Einfügen in eine Liste.

Bei den Methoden first-fit und best-fit ist der freie Speicher im stabilen Zustand (d.h. dynamisches Gleichgewicht zwischen Speicherplatzanforderungen und -freigaben) etwa halb so groß wie der belegte Speicher, d.h. der Speicher ist im Mittel nur etwa zu zwei Dritteln ausgenutzt. Verbesserungen ergibt die Speicherverwaltung gemäß der *Buddy-Methode*, die Aspekte sowohl der Speicherplatzanforderung als auch der Blockverschmelzung berührt.

Die Idee der Buddy-Methode zeigt Abb. 6.63: Der Speicher habe eine Gesamtlänge von $2^{\max}$ (hier: $2^6 = 64$), Anforderungen werden jeweils durch Blöcke der Größe 2^k mit $\min \leq k \leq \max$ befriedigt (hier: $\min = 3$). Die freien und belegten Blöcke einer Länge 2^k werden i.a. jeweils für ein k getrennt in *separaten Listen* verwaltet (aus Darstellungsgründen verzichten wir allerdings hier auf getrennte Frei- und Belegtlisten). Benachbarte Blöcke der Stufe k ergeben einen Block auf Stufe $k + 1$, falls $k < \max$.

Für die *Adressierung* von Buddies ("Kameraden") ist nützlich, daß sich aus der Anfangsadresse eines Blocks leicht wie folgt die Adresse seines Buddies berechnen läßt:

Buddy_k(x) = "Anfangsadresse des Buddies für einen Block der Länge 2^k mit Anfangsadresse x", d.h.
 = $x + 2^k$ falls $(x \equiv 0) \bmod 2^{k+1}$ bzw.
 $x - 2^k$ falls $(x \equiv 2^k) \bmod 2^{k+1}$.

Abb. 6.63 Buddy-Methode

Wenn wir im Beispiel der Abb. 6.63 einen Block der Länge 11 anfordern, gilt wegen $2^3 < 11 < 2^4$, daß wir (mindestens) einen Block der Größe $2^4 = 16$ brauchen. Also geht man die Liste von Blockgröße 2^4 sequentiell durch und testet, ob dort ein freier Block liegt.

Falls ja: Ordne diesen Block zu. Markiere ihn und alle seine Vorgängerknoten bis zur Wurzel als "nicht verfügbar", ebenso den gesamten Unterbaum des zugewiesenen Blocks.

Falls nein: Füge Anforderung in eine Warteschlange ein und prüfe später wieder.

In dem Beispiel würde also der Speicherbereich 16..31 zugewiesen. Eine Speicheranforderung der Größe 25 könnte dagegen nicht erfüllt werden.

Die *Freigabe* eines Blocks erfolgt analog: Ein Block der Größe 2^k wird implizit als frei markiert, d.h. seine Markierung "nicht verfügbar" wird zurückgenommen. Falls auch sein Buddy frei ist, wiederhole das gleiche mit dem Vaterknoten.

Vorteil: Einfache Handhabung.

Nachteile:

- Benachbarte Speicherbereiche, die *nicht Buddies* sind, können *nicht verschmolzen* werden.
- *Interne Fragmentierung*, d.h. Verschwendung von Speicherplatz in den Blöcken, da jeweils $(2^k - \text{benötigte Größe})$ nicht genutzt wird.

Seitenaustauschverfahren

Die Aufgabe eines Seitenaustauschverfahrens besteht im Austauschen von Daten zwischen Hauptspeicher und Hintergrundspeicher. Dafür wird der Speicher in Blöcke gleicher Länge (*Seiten*) 2^m aufgeteilt (2^9 bis 2^{12} Speicherwörter in der Praxis). Ein Zugriff auf eine momentan auf den Hintergrundspeicher ausgelagerte Seite erfordert einen sogenannten *Seitenwechsel*, bei dem die einzulagernde Seite gegen eine nach einer bestimmten Strategie bestimmte Seite des Hauptspeichers ausgetauscht wird. Für eine Speicherverwaltung mit Seitenaustauschverfahren (*virtuelle Speicherverwaltung*) ist eine spezielle Speicheradressierung erforderlich.

Eine Speicheradresse der Länge n hat folgende Bedeutung: $m < n$ Speicherzellen repräsentieren die relative *Adresse auf der Seite* die restlichen $(n-m)$ Speicherzellen stellen die *Seitenadresse* selbst dar.

Die Verwaltung erfolgt über *Seitentabellen*, die für k verwendbare Seiten jeweils Haupt- und Hintergrundadresse angeben.

```
VAR Seitentabelle : ARRAY [1..k] OF
                                RECORD
                                    Hauptadresse      : [0..max];
                                    Hintergrundadresse : ...;
                                END;
```

Die *Hauptadresse* gibt an, daß die entsprechende Seite im Hauptspeicher ab Adresse (Hauptadresse - 1) $\cdot 2^m$ beginnt (falls Hauptadresse = 0, so ist die Seite zur Zeit nicht im Hauptspeicher).

Die *Hintergrundadresse* gibt an, wo die Seite z.B. auf einer Platte steht.

Vorgehen: Es wird ein Programmbefehl oder Datenelement benötigt, welches die Speicheradresse $q \cdot 2^m + r$ besitzt, wobei q die Seitenadresse und r die relative Adresse ist.

(* Programm P6.26 *)

```
IF (q <= 0) OR (q >= k+1) THEN
    "Fehler";
ELSIF Seitentabelle[q].Hauptadresse = 0 THEN
    "Hole Seite der Hintergrundadresse in Hauptspeicher,
    ueberschreibe dabei eine andere Seite, deren
    Hauptadresse := 0 gesetzt wird, nachdem man
    sie ggfs. auf eine Platte gerettet hat.";
ELSE
    "Seite ist bereits im Hauptspeicher
    (also Adresse des Datums leicht zu ermitteln)";
END (* IF *);
```

- Probleme:**
- Wann soll man Seiten austauschen (“vorausschauend” oder erst “auf Anfrage”)?
 - Welche Seite überschreibt man?

Speicherbereinigung

Wenn ein Programm aktiv ist, d.h. zur Laufzeit des Programms, gibt es die in Abb. 6.64 dargestellten “Bereiche” im Speicher:

- *Programmtext* (Code),
- *lokaler* Speicherbereich (*Kellerspeicher*) mit *Zeigervariablen* (p) und
- *dynamischer* Speicherbereich für die Halde (Heap), in den aktuelle Zeiger aus p sowie weitere Zeiger aus dem Heap selbst verweisen.

Abb. 6.64 Differenzierung der einzelnen Speicherbereiche

Die Größe des Codes ist i.a. konstant. Dagegen variiert die Größe des lokalen (hier zur Vereinfachung der Darstellung vernachlässigt) und dynamischen Speicherbereichs während eines Programmlaufs.

Im *Heap* mögen nun “verzeigte” Strukturen (Listen, Bäume, Graphen) aufgebaut worden sein, und es möge ein *Überlauf* drohen, weil die freigewordenen Knoten nicht der Freiliste zugewiesen wurden oder nicht zugewiesen werden konnten, weil gar nicht erkannt wurde, daß sie frei geworden sind.

Nun startet ein *Speicherbereinigungs-Algorithmus* (Garbage Collection), der alle Knoten im Heap, die vom lokalen Speicher und somit vom Programm nicht mehr erreichbar sind, markiert und dann in die Freiliste “*einhängt*”. Statt des Einhängens kann man den Heap auch “*verdichten*”, d.h. auf Kosten der freien Speicherplätze zusammenschieben.

Es gibt zwei grundsätzliche Methoden zur Speicherbereinigung: die *Verweiszählermethode* und die *Markierungsmethode*.

Die für beide Methoden grundlegende Datenstruktur kann man sich als *Graph* mit Knoten, die jeweils *zwei Nachfolger*, einen Inhalt und Hilfsinformation haben, vorstellen. Die Hilfsinformation bei der Verweiszählermethode ist ein *Zähler*, bei der Markierungsmethode eine *Markierung*.

```

TYPE   KPtr   = POINTER TO Knoten;
       Knoten = RECORD
           inhalt      : ...
           (* Zaehler   : CARDINAL;
              fuer Verweiszahlermethode *)
           (* mark      : BOOLEAN;
              fuer Markierungsmethode *)
           nextfrei    : KPtr;
           links, rechts : KPtr;
       END (* RECORD *);

```

In der Realisierung ist KPtr eine Zahl zwischen 0 und M, wenn 1 ... M die Speicheradressen sind und 0 NIL entspricht. Abb. 6.65 zeigt zunächst im unteren Teil ein Beispiel einer Heap-Belegung in grafischer Darstellung:

- Aus dem loc-Bereich verweisen die Zeigervariablen Anker und P in den Heap, in dem es eine Reihe weiterer Verweise zwischen Knoten gibt.
- Eingezeichnet sind jeweils durch gerichtete Kanten die Zeiger links und rechts auf Knoten, falls solche vorhanden sind; andernfalls bezeichnet ein "└" einen NIL-Verweis.
- Die Zeiger nextfrei fehlen hier.

Im oberen Teil der Abb. 6.65 sehen wir die erweiterte, linearisierte Darstellung des Beispiels in der konkreten Belegung eines Heap-Speicherbereiches:

- Beginnend bei Adresse 1 liegen in (zufällig) lexikographischer Reihenfolge die Knoten A bis I (Heap-Ende bei Speicherplatz 110).
- Neben dem eigentlichen Inhalt (jeweils durch die Knotenbezeichnung "A" bis "I" repräsentiert) zeigt die Abbildung die drei Verweise pro Knoten: auf den linken bzw. rechten Knoten sowie - falls dazu gehörig - den Nachfolger in der Freiliste.

Verweiszählermethode

Jeder Knoten enthält einen *Zähler*, der angibt, wie oft auf ihn verwiesen wird. Ist der Zähler null, dann kann der Knoten in die Freiliste eingehängt werden.

Nachteile: Es sind keine zyklischen Strukturen erfaßbar. Bei jeder Freigabe müssen die Zähler der Nachfolgeknoten ebenfalls erniedrigt werden.

Wegen dieser Nachteile wird i.a. die Markierungsmethode verwendet.

Abb. 6.65 Beispiel einer Heap-Belegung

Markierungsmethode

Jeder Knoten enthält eine *Markierung*, die angibt, ob der Knoten vom Kellerspeicher aus erreichbar ist. Zunächst werden alle Knoten markiert, die direkt erreichbar sind. In einem *iterativen* Verfahren werden dann alle Knoten markiert, die nur indirekt, d.h. nur über andere erreichbare Knoten (z.B. über den Vorgänger in einer verketteten Liste) erreichbar sind. Diese Methode hat Ähnlichkeit mit dem früher vorgestellten Algorithmus zu Bestimmung der *starken Zusammenhangskomponenten* in gerichteten Graphen.

Sei k die Länge eines Knotens im Speicher (k = Zahl benötigter Speicherplätze) und 0 die erste und $M-1$ die letzte Adresse im Speicher. Programm P6.27a zeigt in Pseudocode den grundlegenden Algorithmus der Markierungsmethode.

```

i := 0;                                (* Programm P6.27a *)
WHILE i <= M-1 DO
  "Markiere Knoten mit Adresse i als FALSE";
  INC (i, k);                          (* Knoten sind (noch) nicht erreichbar *)
END (* WHILE *);
"Markiere alle direkt vom lokalen Speicher erreichbaren
Knoten mit TRUE";                      (* Knoten sind nicht erreichbar *)
i := 0;
WHILE i <= M-1 DO
  j := i + k;
  IF "Knoten mit Adresse i besitzt keine Nachfolger"
    (* d.h. links = NIL und rechts = NIL *)
    OR "dieser Knoten ist mit FALSE markiert" THEN
    i := j;
  ELSE
    IF (links # NIL)
      AND "Knoten links^ mit FALSE markiert" THEN
      "markiere Knoten links^ mit TRUE";
      "j := Minimum (j, Adresse von Knoten links^)";
    END (* IF *);
    IF (rechts # NIL)
      AND "Knoten rechts^ mit FALSE markiert" THEN
      "markiere Knoten rechts^ mit TRUE";
      "j := Minimum (j, Adresse von Knoten rechts^)";
    END (* IF *);
    i := j;
  END (* IF *);
END (* WHILE *);
"Haenge alle FALSE-markierten Knoten in die Freiliste ein."

```

Hat man noch Speicherplatz für den Speicherbereinigungs-Algorithmus zur Verfügung, dann kann man - statt des dauernden zeitaufwendigen Zurückspringens im Speicher - den nachfolgenden Algorithmus des Programms P6.27b benutzen. Dazu wird der in Abschnitt 3.2.4.2 vorgestellte *Stack* verwendet.

```

(* Programm P6.27b *)
REPEAT
  WHILE NOT Empty (S) AND "(Top (S) hat keine Nachfolger)" DO
    Pop (S);
  END (* WHILE *);
  IF NOT Empty (S) THEN
    p := Top (S); Pop(s);
    IF (p^.links # NIL) AND NOT p^.links^.mark THEN
      p^.links^.mark := TRUE;
      Push (S, p^.links);
    END (* IF *);
    IF (p^.rechts # NIL) AND NOT p^.rechts^.mark THEN
      p^.rechts^.mark := TRUE;
      Push (S, p^.rechts);
    END (* IF *);
  END (* IF *);
UNTIL Empty (S);
"Haenge alle FALSE-markierte Knoten in die Freilliste ein."

```

Analyse: Wenn am Ende n Knoten mit TRUE markiert sind und $m = M/k$ die Zahl der maximal speicherbaren Knoten ist, dann arbeitet der Algorithmus von P6.27a mit $O(n-m)$ Zeit und $O(1)$ Platz, der Algorithmus von P6.27b mit $O(n+m)$ Zeit und $O(m)$ Platz.

Idee: Kombiniere beide Algorithmen zu einem Algorithmus, der mit einem lngenbeschrnkten Stack gem P6.27b arbeitet bis ein berlauf stattfindet, und dann auf P6.27a solange "umsteigt", bis wieder Platz im Stack ist.

Generelle Probleme bei der Speicherbereinigung:

- Je *weniger* Speicherplatz freigegeben werden kann, desto *lnger* laufen die Algorithmen.
- Wartet man lange mit der Speicherbereinigung, so entsteht allmhlich eine *Fragmentierung*. Daher sollte man frhzeitig den Garbage-Collector aufrufen und freie Blocke verschmelzen.
- Das Zusammenschieben ist i.a. wegen der hufig komplexen "Verzeigerung" der Knoten sehr aufwendig.
- Der Garbage-Collector hat i.a. keinen freien Speicherplatz.

6.6.2 Implementierung von Stacks

In diesem Abschnitt werden Konzepte fr die Implementierung von mehreren *Stacks* (aus Grunden der Anschaulichkeit ber dem gleichen Datentyp) vorgestellt, die aufgrund dynamischen Wachsens - oder auch Schrumpfens - um Speicherplatz konkurrieren.

Ein einfacher Algorithmus könnte ein ein-dimensionales Array zum Speichern der Stacks vorsehen und Anfang und Ende des i -ten Stacks in Variablen **POINTER** [i] (zeigt auf das oberste Stack-Element) und **BASE** [i] (zeigt auf das Element *vor* dem untersten Stack-Element des i -ten Stacks) notieren. Falls nun im i -ten Stack durch Speicheranforderung ein *Überlauf* (*Overflow*) droht, d.h. es gilt bereits **POINTER** [i] = **BASE** [$i+1$], dann muß etwa durch folgende Prozedur overflow Speicherplatz beschafft werden.

```

Prozedur overflow (i : [1 .. n]):    (* Overflow in Stack i *)
Suche das kleinste j mit POINTER [j] < BASE [j + 1].
Falls es ein solches j nicht gibt,
    dann ist das Array vollstaendig gefuehlt, und es muss mit
    Overflow-Error abgebrochen werden;
anderenfalls (d.h. es existiert ein solches j),
    falls i < j ist, dann
        verschiebe die Stacks j, j - 1, ..., i + 1 um jeweils
        einen Platz nach oben und inkrementiere entsprechend
        POINTER und BASE dieser Indizes um 1;
    anderenfalls (d.h. falls j < i)
        verschiebe die Stacks j + 1, ..., i - 1, i um jeweil
        einen Platz nach unten und dekrementiere entsprechend
        deren POINTER- und BASE-Werte um 1.

```

Nachteil:

Wenn anschließend wieder ein Overflow (i) auftritt, dann muß man den Overflow-Algorithmus schon wieder starten. Besser wäre es, im Overflow-Fall *alle* Stacks entsprechend ihrer Größe und/oder ihres bisherigen Wachstums neu zu verteilen. Einen solchen Ansatz verfolgt der nachfolgende *Garwick-Algorithmus*.

Garwick-Algorithmus

Der Garwick-Algorithmus eignet sich zur Overflow-Behandlung von n (längenbeschränkten) Stacks, Tabellen, Schlangen usw.

Bei *Overflow* eines Stacks wird der Speicherplatz an alle n Stacks neu verteilt, und zwar werden z.B. 10% des gerade freien Speicherplatzes gleichmäßig *allen* Stacks zugewiesen und 90% des freien Speicherplatzes werden *proportional* zum jeweiligen *Zuwachs* seit dem letzten Overflow an die einzelnen Stacks verteilt (auch andere prozentuale Verteilungen sind natürlich denkbar).

Abb. 6.66 zeigt ein Beispiel zur Anwendung des Garwick-Algorithmus: **POINTER** und **BASE** geben die aktuelle Situation an. Der Befehl **PUSH** (2, x) bewirkt in der gezeigten Situation einen Overflow, da **POINTER** [2] = **BASE** [3] ist. **OLD-POINTER** gibt die Position der Zeiger auf das oberste Stack-Element nach der letzten Overflow-Behandlung an.

Abb. 6.66 Garwick-Algorithmus zur Implementierung von Stacks

In der in Abb. 6.66 gezeigten Situation standen nach der letzten Overflow-Behandlung

- für Stack 1 der Bereich von Zelle 1 bis Zelle 9 zur Verfügung, und es wurde der Bereich von Zelle 1 bis Zelle 6 belegt;
- für Stack 2 der Bereich von Zelle 10 bis Zelle 17 zur Verfügung, und es wurde der Bereich von Zelle 10 bis Zelle 15 belegt usw.

Die neue Situation, bei der der Overflow durch PUSH(2, x) entstand, bedeutet aktuell

- für Stack 1 eine Abnahme des Speicherplatzes (um 2 Zellen),
- für Stack 2 einen Zuwachs des Speicherplatzes (um 2 Zellen),
- für Stack 3 eine Abnahme (um 3 Zellen),
- für Stack 4 eine Abnahme (um 2 Zellen; Stack 4 ist gerade leer),
- für Stack 5 einen Zuwachs (um 1 Zelle).

Bei der Berechnung der neuen Basiswerte NEWBASE und der Neudefinition der POINTER-Werte werden vom verfügbaren freien Speicherplatz (Zellen 5, 6, 7, 8, 9, 21, 22, 23, 24, 25, 26, 27, 28, 29, 32, also 15 freie Plätze) 10% gleichmäßig an alle fünf Stacks verteilt (im Beispiel erhält jeder Stack somit $0,1 \cdot 15/5$, also 0 Speicherplätze aus diesem Kontingent) und 90% gemäß Wachstum.

Im Beispiel erhalten also

Stack 2 $(0,9 \cdot 15 \cdot 2) / 3 = 9$ Speicherplätze und

Stack 5 $(0,9 \cdot 15 \cdot 1) / 3 = 4,5$, also abgerundet 4 Speicherplätze.

Der Garwick-Algorithmus liefert dann für das Beispiel aus Abb. 6.66 folgende Werte für POINTER und NEWBASE:

4	12	24	24	26	0	4	21	24	24	32
1	2	3	4	5	1	2	3	4	5	6
POINTER					NEWBASE					

Abb. 6.67 POINTER- und NEWBASE-Array des Garwick-Algorithmus

NEWBASE wird nach BASE, POINTER nach OLDPOINTER kopiert, dann wird zur Anweisung PUSH (2, x) zurückverzweigt.

Eine Realisierung des Garwick-Algorithmus zeigt Programm P6.28.

(* Programm P6.28 *)

```

TYPE speicher_fuer_stacks = RECORD
    PTR, OLDPOINTER : ARRAY [1..n] OF [0..M+1];
    BASE, NEWBASE   : ARRAY [1..n+1] OF [0..M];
    DELTA           : ARRAY [1..n] OF [0..M];
    SPEICHER        : ARRAY [1..M] OF D;
END (* RECORD *);
(* DELTA ist ein Hilfsfeld zur Berechnung der Zuwachse;
   M gibt global die Groesse des Speichers für die n Stacks an. *)

```

```

PROCEDURE Overflow;

```

```

(* Wir arbeiten m.einer Variablen d. Typs speicher_fuer_stacks *)

```

```

VAR v          : REAL;
    u          : INTEGER;
    sum, incre : CARDINAL;
    j          : [1..n];
BEGIN (* Overflow *)
    sum := M;   incre := 0;
    FOR j := 1 TO n DO (* Berechnung d. freien Speicherplatzes *)
        sum := sum - PTR [j] + BASE [j];
        IF PTR [j] > OLDPOINTER [j] THEN (* Stack j hatte Zuwachs *)
            DELTA [j] := PTR [j] - OLDPOINTER [j];
            INC(incre, DELTA [j]);        (* Berechnung des
                                           gesamten Zuwachses *)
        ELSE
            DELTA [j] := 0;                (* kein Zuwachs *)
        END (* IF *);
    END (* FOR j *);
    IF sum <= 0 THEN (* 1 *)
        (* Fehlerabbruch; kein Speicher mehr frei *);
    ELSE
        (* freien Platz sum zu 10% auf alle Stacks neu verteilen;
           u : Anteil pro Stack an 10% *)
        u := TRUNC (0.1 * FLOAT (sum) / FLOAT (n) + 0.5);
        IF u * n >= sum THEN (* sum bereits aufgebraucht *)
            DEC(u);          (* bringt wieder "Spielraum" *)
        END (* IF *);
        (* Rest ( ≈ 90%) anteilig nach Zuwachs verteilen;
           v : Zuwachsfaktor für 1 Speicherplatz *)
        v := FLOAT (sum - u * n) / FLOAT(incre); (* 2 *)
        NEWBASE [1] := 0;   NEWBASE [n+1] := M;
        FOR j := 2 TO n DO
            NEWBASE [j] := NEWBASE [j-1]
                + PTR [j-1] - BASE [j-1]
                + u
                + TRUNC (FLOAT (DELTA [j-1]) * v);
            (* Beginn Stack j-1
               + Laenge von stack j-1
               + u-Anteil für jeden Stack
               + v-Anteil für Wachstum für Stack j-1 *)
        END (* FOR j *);
        Speicherumordnen;

```



```

    FOR j := 1 TO n DO
        OLDPOINTER [j] := PTR [j];
    END (* FOR j *);
END (* FOR *);
END Overflow;

PROCEDURE Speicherumordnen;
VAR p      : [2..n];
    j, k : [2..n+1];
BEGIN (* Speicherumordnen *)
    j := 2;
    WHILE j <= n DO
        k := j;
        IF NEWBASE [k] < BASE [k] THEN
            Verschieben (k);
        ELSE
            WHILE NEWBASE [k+1] > BASE [k+1] DO
                INC(k);
            END (* WHILE *);
            FOR p := k TO j BY -1 DO
                Verschieben (p);
            END (* FOR *);
        END (* IF *);
        j := k + 1;
    END (* WHILE *);
END Speicherumordnen;

PROCEDURE Verschieben (p : [2..n]);
VAR t : [0..M+1];
    s : INTEGER;
BEGIN (* Verschieben *)
    s := NEWBASE [p] - BASE [p];
    (* s gibt an, um wieviel Stellen p verschoben werden muss *)
    IF s # 0 THEN
        IF s > 0 THEN
            FOR t := PTR [p] TO (BASE [p] + 1) BY -1 DO
                SPEICHER [t+s] := SPEICHER [t];
            END (* FOR *);
        ELSE
            FOR t := (BASE [p] + 1) TO PTR [p] DO
                SPEICHER [t+s] := SPEICHER [t];
            END (* FOR *);
        END (* IF *);
    END (* IF *);
END Verschieben;

```

Bemerkungen

1. An obigem Beispiel erkennt man, daß der Algorithmus nicht korrekt arbeitet. Wird beim nächsten Mal z.B. Overflow wegen PUSH (4, x) gerufen, so würde kein zusätzlicher Speicherplatz für Stack 4 zur Verfügung gestellt, falls sich $u = 0$ ergibt! Nach dem Zurückverzweigen würde also x fälschlicherweise dem Stack 5 als unterstes Element zugeordnet.

Dies kann man durch eine leichte Modifikation der Prozedur Overflow korrigieren:

Entweder übergibt man die Nummer i des Stacks, die den Overflow bewirkt hatte und setzt an der Stelle (*1*) im Algorithmus DELTA $\text{DELTA}[i] := \text{DELTA}[i] + 1$ (dann aber incr mit 1 initialisieren!), oder man fängt den Fall $u = 0$ an der Stelle (*2*) im Algorithmus ab, indem man durch Abändern von u und v sicherstellt, daß jedem Stack mindestens *ein* freier Speicherplatz zugewiesen werden muß.

2. Der Overflow-Algorithmus wird sehr häufig aufgerufen, wenn der Speicher fast voll ist und ohnehin bald ein Abbruch wegen Platzmangel eintreten wird. Daher sollte man den Algorithmus nicht erst bei $\text{sum} \leq 0$ endgültig abbrechen, sondern bereits bei $\text{sum} > \text{mfs}$ ("minimaler freier Speicherplatz"), wobei "mfs" eine vom Programmierer vorgegebene Größe ist, und gegebenenfalls extern neuen Speicher besorgen (Empfehlung für $\text{mfs} \approx M/8$.)

Der Garwick-Algorithmus ist in den Fällen effizient, in denen nur etwa 50% des Speicherplatzes durch die Stacks belegt sind.

Anhang A: Mathematische Grundbegriffe und Formeln

Kartesisches Produkt, Tupel, Relation, Eigenschaften von Relationen und Abbildungen

Seien $A_1, \dots, A_n$ Mengen.

Die Menge $A_1 \times A_2 \times \dots \times A_n = \{(a_1, a_2, \dots, a_n) \mid (a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n)\}$ heißt *kartesisches Produkt*. Die Elemente aus $A_1 \times A_2 \times \dots \times A_n$ heißen *Tupel* oder auch *n-Tupel*.

Eine Teilmenge $R \subseteq A_1 \times \dots \times A_n$ heißt *n-stellige Relation* (auf $A_1, \dots, A_n$).

$R \subseteq A \times B$ sei eine zweistellige Relation.

(a) R heißt *Abbildung* oder *Funktion* von A nach B (geschrieben $R: A \rightarrow B$) : $\Leftrightarrow$

$$\forall a \in A, b, b' \in B : (a, b) \in R \wedge (a, b') \in R \Rightarrow b = b'.$$

(b) $R^{-1} \subseteq B \times A$ heißt die zu R *inverse Relation*,

$R^{-1}(B) := \{a \mid \exists b: (a, b) \in R\}$ heißt *Definitionsbereich* von R und

$R(A) := \{b \mid \exists a: (a, b) \in R\}$ heißt *Wertebereich* von R .

(c) Eine Abbildung $R: A \rightarrow B$ heißt *total* : $\Leftrightarrow R^{-1}(B) = A$.

Eine Abbildung $R: A \rightarrow B$ heißt *partiell* : $\Leftrightarrow R^{-1}(B) \subset A$.

(d) Eine totale Abbildung $R: A \rightarrow B$ heißt

surjektiv : $\Leftrightarrow R(A) = B$, d.h. $\forall b \in B \exists a \in A: (a, b) \in R$,

injektiv : $\Leftrightarrow \forall a, a' \in A, b \in B : (a, b) \in R \wedge (a', b) \in R \Rightarrow a = a'$

bijektiv : $\Leftrightarrow R$ ist surjektiv und injektiv.

$R \subseteq A \times A$ sei eine zweistellige Relation. R heißt

reflexiv : $\Leftrightarrow \forall a \in A: (a, a) \in R$,

irreflexiv : $\Leftrightarrow \forall a \in A: (a, a) \notin R$,

<i>symmetrisch</i>	$:\Leftrightarrow$	$\forall a, a' \in A: (a, a') \in R \Rightarrow (a', a) \in R,$
<i>antisymmetrisch</i>	$:\Leftrightarrow$	$\forall a, a' \in A: (a, a') \in R \wedge (a', a) \in R \Rightarrow a=a',$
<i>asymmetrisch</i>	$:\Leftrightarrow$	$\forall a, a' \in A: (a, a') \in R \Rightarrow (a', a) \notin R,$
<i>transitiv</i>	$:\Leftrightarrow$	$\forall a, a', a'' \in A: (a, a') \in R \wedge (a', a'') \in R$ $\Rightarrow (a, a'') \in R,$
<i>semikonnex</i>	$:\Leftrightarrow$	$\forall a, a' \in A: (a=a' \vee (a, a') \in R \vee (a', a) \in R).$
<i>Äquivalenzrelation</i>	$:\Leftrightarrow$	R ist reflexiv, symmetrisch und transitiv.
<i>partielle Ordnung</i>	$:\Leftrightarrow$	R ist reflexiv, antisymmetrisch und transitiv,
<i>lineare Ordnung</i>	$:\Leftrightarrow$	R ist partielle Ordnung und semikonvex,
<i>Halbordnung</i>	$:\Leftrightarrow$	R ist partielle Ordnung.

Sei B eine Menge und $A \subseteq B$. Dann heißt die totale Abbildung $C_A : B \rightarrow \{0, 1\}$ mit

$$C_A(b) := \begin{cases} 1, & \text{falls } b \in A \text{ bzw.} \\ 0, & \text{falls } b \notin A \end{cases}$$

charakteristische Funktion der Menge A .

Fakultät, Stirlingsche Formel

$$n! := \prod_{i=1}^n i \quad (n \geq 0, \text{ speziell } 0! = 1)$$

$$n! = (n-1)! \cdot n$$

$$n! \approx \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \frac{1}{12n}\right) \text{ mit } e \approx 2.718282$$

Binomialkoeffizienten, Binomischer Lehrsatz

Seien $n, k \in \mathbb{N}_0$:

$$\binom{n}{k} := \begin{cases} \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{k \cdot (k-1) \cdot \dots \cdot 1} & \text{für } n \geq k \geq 0 \\ 0 & \text{für } k > n \geq 0 \end{cases}$$

Es gilt für alle $n \in \mathbb{N}_0, k \in \mathbb{N}$:

$$\binom{n}{k} = \frac{n!}{(n-k)! k!} \quad \text{speziell} \quad \binom{n}{0} = 1$$

$$\binom{n}{k} = \binom{n}{n-k}$$

$$\binom{n}{k} = \frac{n}{k} \cdot \binom{n-1}{k-1}$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{weil:} \quad \binom{n}{k} = \frac{n}{k} \cdot \binom{n-1}{k-1} = \left(1 + \frac{n-k}{k}\right) \cdot \binom{n-1}{k-1}$$

$$\binom{n+1}{k+1} = \sum_{i=k}^n \binom{i}{k}$$

$$\sum_{k=0}^m \binom{n+k}{k} = \binom{m+n+1}{m}$$

$$(x+y)^n = \sum_{i \in \mathbb{N}} \binom{n}{i} x^i y^{n-i} = \sum_{i=0}^n \binom{n}{i} x^i y^{n-i} \quad (\text{speziell gilt: } 0^0 = (1-1)^0 = 1)$$

Summenformeln

$$\sum_{i=0}^n i = \sum_{i=0}^n \binom{i}{1} = \frac{1}{2} n \cdot (n+1)$$

$$\sum_{i=0}^n i^2 = \sum_{i=0}^n \left(2 \cdot \binom{i}{2} + \binom{i}{1}\right) = \binom{n+1}{2} = \dots = \frac{1}{6} n \cdot (n+1) \cdot (2n+1)$$

$$\begin{aligned} \sum_{i=0}^n i^3 &= \sum_{i=0}^n \left(6 \cdot \binom{i}{3} + 6 \cdot \binom{i}{2} + \binom{i}{1}\right) = \frac{1}{4} n \cdot (n+1) \cdot (n^2+n) \\ &= \frac{1}{4} n^2 \cdot (n+1)^2 \end{aligned}$$

Harmonische Reihe

Seien $\ln$ der natürlicher Logarithmus und
 C die Eulersche Konstante ($\approx 0.577\,216$).

$$H_k := \sum_{i=1}^k \frac{1}{i} \quad (\text{für } k \geq 0; \text{ speziell: } H_0 = 0)$$

$$H_n = \ln(n) + C + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon \quad \text{mit } 0 < \varepsilon < \frac{1}{252n^6}$$

$$\sum_{i=1}^n H_i = (n+1) \cdot H_n - n$$

Anhang B: Syntaxdiagramme für MODULA-2

1. Die auf den nächsten Seiten folgende Darstellung der Syntax von MODULA-2 in Form von *Syntaxdiagrammen* (zu Beginn von Kapitel 2 eingeführt) stützt sich auf das Buch von *Dal Cin, Lutz und Risse* "Programmierung in Modula-2" aus dem Teubner-Verlag. Sie wurde bezüglich der *EBNF-Darstellung* (ebenfalls zu Beginn von Kapitel 2 vorgestellt) im Buch "Programmieren in Modula-2" von *Wirth*, Springer-Verlag, geprüft.
2. Für die Darstellung der kontextfreien Syntax gelten die üblichen, nicht formal definierten *Einschränkungen* (beispielsweise der Ausschluß reservierter Wörter als Bezeichner).
3. Die *Meta-Syntax* der Darstellung ist gegenüber vergleichbaren Darstellungen in zwei Punkten erweitert:
 - Zur besseren Übersicht sind die Produktionen *durchnummeriert*. Die Nummern sind (als Verweise) auch bei den Nichtterminalsymbolen in den Diagrammen angegeben. Formal sind sie bedeutungslos.
 - Grundsätzlich sind (bedeutungslose) *Leerzeichen* (Blanks, Spaces) überall erlaubt, wo in den Diagrammen Pfeile (Verbindungslinien) stehen. Sie sind notwendig, wo die Gliederung sonst nicht eindeutig ist (beispielsweise, wenn einem Bezeichner ein Wortsymbol folgt). Wo Leerzeichen ausgeschlossen sind (oder Bedeutung tragen), wurden die Verbindungen in den Diagrammen *fett* gezeichnet. *Zeilenwechsel* und *Kommentare* werden wie Leerzeichen behandelt.
4. Die durch die Syntaxdiagramme repräsentierte *Grammatik* hat drei Startsymbole (Wurzeln), nämlich ImplementationModule (70), DefinitionModule (67) und ProgramModule (11) und ist damit im strengen Sinne keine richtige Grammatik (die nur *ein* Startsymbol haben darf).
5. Auf der nächsten Seite folgt ein Index zum schnelleren Auffinden gesuchter Syntaxdiagramme.

Index zu den Syntaxdiagrammen

ArrayType	19	FormalParameters	57	Real	8
Assignment	46	FormalType	59	RecordType	20
Block	12	FormalTypeList	63	Relation	28
CaseLabelList	24	ForStatement	51	RepeatStatement	50
CaseStatement	48	hexDigit	7b	ReturnStatement	61
character	10	Ident	1	Set	43
ConstDeclaration	26	IfStatement	47	SetType	17
ConstElement	34	ImplementationModule	70	SimpleConstExpr	29
ConstExpr	27	Import	65	SimpleExpression	40
ConstFactor	32	Integer	4	SimpleType	18
ConstSet	33	letter	5	Statement	45
ConstTerm	30	LoopStatement	52	StatementSequence	14
Declaration	13	ModuleDeclaration	64	String	9
Definition	68	MulOperator	31	SubrangeType	16
DefinitionModule	67	Number	3	Term	41
Designator	38	octDigit	7a	Type	36
digit	6	ParamSection	58	TypeDeclaration	35
Element	44	PointerType	25	TypeDefinition	69
Enumeration	15	ProcedureCall	60	VariableDeclaration	37
ExitStatement	53	ProcedureDeclaration	55	Variant	23
Export	66	ProcedureHeading	56	VariantFieldList	22
Expression	39	ProcedureType	62	WhileStatement	49
Factor	42	ProgramModule	11	WithStatement	54
FieldList	21	QualIdent	2

String₉ character₁₀ ProgramModule₁₁ Block₁₂

SetType₁₇ SimpleType₁₈ ArrayType₁₉ RecordType₂₀ FieldList₂₁

VariantFieldList₂₂ Variant₂₃ CaseLabelList₂₄ PointerType₂₅ ConstDeclaration₂₆

ConstExpr₂₇ Relation₂₈ SimpleConstExpr₂₉ ConstTerm₃₀ MulOperator₃₁

VariableDeclaration₃₇ Designator₃₈

Expression₃₉ SimpleExpression₄₀ Term₄₁

IfStatement₄₇ CaseStatement₄₈ WhileStatement₄₉

RepeatStatement₅₀ ForStatement₅₁ LoopStatement₅₂ ExitStatement₅₃ WithStatement₅₄ ProcedureDeclaration₅₅

ProcedureHeading₅₆ FormalParameters₅₇ ParamSection₅₈ FormalType₅₉ ProcedureCall₆₀ ReturnStatement₆₁

TypeDefiniton₆₉ ImplementationModule₇₀

Literatur

Folgende einführende, vorrangig deutschsprachige Informatik-Literatur sei für ein ergänzendes und das vorliegende Buch relativierendes Studium empfohlen.

Abelson, H., Sussman, G.J.: *Structure and Interpretation of Computer Programmes*, MIT Press, 1985.

Aho, A., Hopcroft, J., Ullmann, J.: *Data Structures and Algorithms*, Addison-Wesley, 1985.

Bauer, F.L., Goos, G.: *Informatik - eine einführende Übersicht*, Springer, 1990.

Bauer, F.L., Wössner, H.: *Algorithmische Sprache und Programmentwicklung*, Springer, 1984.

Bauknecht, K., Zehnder, C.A.: *Grundzüge der Datenverarbeitung*, Teubner, 1989.

Claus, V., Schwill, A.: *Duden Informatik, ein Sachlexikon für Studium und Praxis*, BI Dudenverlag, 1989.

Dijkstra, E.: *A Discipline of Programming*, Prentice Hall, 1976.

Frühauf, K., Ludewig, J., Sandmayr, H.: *Software-Projektmanagement und -Qualitätssicherung*, Teubner, 1991.

Gries, D.: *The Science of Programming*, Springer, 1981.

Goldschlager, L., Lister, A.: *Informatik - eine moderne Einführung*, Hanser, 1986.

Hotz, G.: *Einführung in die Informatik*, Teubner, 1990.

Klaeren, H.: *Vom Problem zum Programm*, Teubner, 1990.

Knuth, D.A.: *The Art of Computer Programming, Vol. 1-3*, Addison-Wesley, 1973.

Loeckx, J., Mehlhorn, K., Wilhelm, R.: *Grundlagen der Programmiersprachen*, Teubner, 1986.

Ludewig, J.: *Sprachen für die Programmierung - eine Übersicht*, BI, 1985.

Mehlhorn, K.: *Datenstrukturen und effiziente Algorithmen, Band 1 Sortieren und Suchen*, Teubner, 1986.

Nievergelt, J., Hinrichs, K.: *Programmierung und Datenstrukturen*, Springer, 1986.

Noltemeier, H.: *Informatik I - III*, Hanser, 1981-89.

Ottmann, T., Widmeyer, P.: *Algorithmen und Datenstrukturen*, BI, 1990.

Pomberger, G.: *Softwaretechnik und MODULA-2*, Hanser, 1984.

Randell, B. (Hrsg.): *The Origins of Digital Computers*, Springer, 1982.

Rembold, U.: *Einführung in die Informatik*, Hanser, 1987.

Schabach, R.: *Grundlagen der Informatik*, Vieweg, 1988.

Schefe, P.: *Informatik - Eine konstruktive Einführung*, BI, 1987.

Sedgewick, R.: *Algorithmen*, Addison-Wesley, 1990.

Waldschmidt, E.H.: *Informatik für Ingenieure*, Oldenbourg, 1987.

Waldschmidt, E.H., Walter, H.K.-G.: *Grundzüge der Informatik*, BI, 1990.

Wirth, N.: *Algorithmen und Datenstrukturen mit Modula-2*, Teubner, 1986.

Wulf, W., Shaw, M., Hilfinger, P., Flon, L.: *Fundamental Structures of Computer Science*, Addison-Wesley, 1981.

Für eine Vertiefung der MODULA-2-Kenntnisse bieten sich u.a. die folgenden Bücher an.

Blaschek, G., Pomberger, G., Ritzinger, F.: *Einführung in die Programmierung mit MODULA-2*, Springer, 1986.

Dal Cin, M., Lutz, J., Risse, T.: *Programmierung in MODULA-2*, Teubner, 1986.

Pudlatz, H.: *Einführung in die Programmiersprache Modula-2*, Vieweg, 1988.

Sale, A.: *Modula 2 - Durch systematischen Entwurf zum korrekten Programm*, Addison-Wesley, 1987.

Wirth, N.: *Programmieren in MODULA-2*, Springer, 1991.

Abkürzungsverzeichnis

ADT	Abstrakter Datentyp
ALU	Arithmetic Logical Unit (Rechenwerk)
ave	average (durchschnittlicher Fall bei Aufwandsabschätzung)
A^*	freies Monoid über einem Alphabet A
BNF	Backus-Naur-Form
CPU	Central Processing Unit (Rechenwerk und Steuerwerk)
DB	Datenbank
EBNF	erweiterte Backus-Naur-Form
LSK	Lese-Schreibkopf (einer Turing-Maschine)
max	maximum (schlechtester Fall bei Aufwandsabschätzung)
min	minimum (günstigster Fall bei Aufwandsabschätzung)
$\mathbb{N}$	Menge der natürlichen Zahlen ohne null
$\mathbb{N}_0$	Menge der natürlichen Zahlen mit null
$O(\dots)$	Zeitkomplexitätsklasse (obere Schranke)
RAM	Random Access Memory (wahlfreier Zugriff)
TM	Turing-Maschine
wp	weakest precondition (schwächste Vorbedingung)
$\mathbb{Z}$	Menge der ganzen Zahlen

Index

A

Abbildung 413
Ableitungsbaum 38
Ableitung 26
Abstrakter Datentyp 143; 157; 172
Abstraktion 101; 107; 155; 196; 253
Addierwerk 32
Adjazenzmatrix 270
Akkumulator 32
Aktionsteil einer Regel 250
Algorithmus 9
ALLOCATE 128
Alphabet 22
Alternative 70; 206
Angewandte Informatik 45
Anker 129; 133; 139
antisymmetrisch 414
Anweisungsüberdeckung 231
Anweisung 58; 61; 72
Anwendungen der Informatik 45
Äquivalenzproblem 18
Äquivalenzrelation 414
Array 111
Assembler 28; 156
asymmetrisch 414
asymptotische Abschätzung 258
Aufwand des Sortierens 365; 388
Aufzählungstyp 75
Ausdruck 62; 72
Ausführungspunkt 89
Ausführungsreihenfolge 162
Aus-/Eingabe 148
ausgeglichener Baum 276
ausgeglichener Suchbaum 309
Auswahl 226

Auswirkungen der Informatik 48
average case 369
AVL-Baum 310
axiomatische Semantik 195
azyklisch 267

B

B*-Baum 337
B-Baum 332
Backus-Naur-Form 25; 51
Balancefaktor 310
balancierter Baum 340
Band (einer TM) 12
Basistyp 108
Baum 275; 302; 310; 326; 331; 340
bedingte Anweisung 69; 206
Bedingungsüberdeckung 231; 233
Befehlsregister 30
berechenbare Funktion 17
Berechenbarkeit 9; 16
Bereichstyp 75
best case 369
best-fit 398
Betriebssystem 36, 397
Bezeichner 55; 61; 71
Binärbaum 276; 339; 374
binärer Suchbaum 302
binäres Suchen 289
Binden 161
Bindestrich-Informatik 45
Binomialkoeffizient 414
Bit 33
BITSET 109
Black-Box-Test 227; 232
Blatt 276
Blindelement 141
Block 58; 61; 65; 397
Blockverschmelzung 397; 399

Blockverwaltung 398
 BNF 52
 BOOLEAN 74
 Boyer-Moore (BM) 295
 Bruder 276; 337
 Bubblesort 372
 Bucketsort 389
 Buddy-Methode 399
 Byte 33; 153

C

call-by-reference 86
 call-by-value 86
 CARDINAL 73
 CHAR 74
 charakteristische Funktion 262; 414
 Churchsche These 18
 Cliquesproblem 269
 Compiler 28; 38

D

Datei 146; 148; 152
 Datenabstraktion 156
 Datenkapselung 165
 Datentyp 72; 107; 172
 DEALLOCATE 128
 Decodierer 30
 Decompiler 38
 denotationale Semantik 194; 196
 Dereferenzieren 138
 Dereferenzieroperator 129
 Determiniertheit 10
 Determinismus 10
 deterministisch 10; 259
 Dezimalsystem 22
 Digitale Speicher 33
 Direktes Aussuchen 369
 Direktes Einfügen 370
 Direktes Mischen 394
 Diskriminatorfeld 123
 DISPOSE 128
 doppelt verkettete Liste 141

Dualsystem 22
 dynamische Variablen 127
 dynamischer Speicherbereich 402
 dynamisches Hashing 343

E

EBNF 52
 Editor 39
 Effektivität 83
 Effizienz 11; 83; 255
 einfache Sortiervverfahren 369
 Ein-/Ausgabe 148
 Eingabevariablen 76
 elementare Datentypen 72
 Endknoten 267
 Entscheidungsbaum 365
 erfolglose Suche 326
 erfolgreiche Suche 326
 Erkennungssystem 28
 Erzeugungssystem 28
 Eulersche Konstante 416
 Eulerscher Weg 269
 exakte Schranke 259
 exponentiell-zeitbeschränkt 260
 Export 163
 externe Sortiervverfahren 368; 393
 externer Speicher 34; 146
 externes Sortieren 361

F

Fädelung 282
 Fakultät 414
 Fallunterscheidung 66
 Family-order-sequentiell 284
 Färbungsproblem 269
 Fehler 205; 222
 Fehlstand 363
 Feld 111
 Fibonacci-Bäume 323
 Fibonacci-Zahlen 83; 323; 396
 FIFO-Prinzip 170
 File 146

File-Operationen 149
Finitheit 10
first-fit 398
Folge 146
FOR-Schleife 95
formatierte Ein-/Ausgabe 148
Fragmentierung 400; 406
Funktion 70; 156; 413
funktional-rekursiv 83; 97
funktionale Sprachen 235; 241
funktionale Programme 55
Funktionsprozedur 185
Funktionswert 94

G

Garbage Collection 129; 402
Garwick-Algorithmus 407
gerichteter Graph 266
Gewicht 269
gewichtsbalancierter Baum 341
Glass-Box-Test 227; 233
global 94
GOTO 101
Grammatik 9; 22
Grad 276
Graph 265
Grundtyp 75; 146
Gültigkeitsbereich 87; 94; 119; 125

H

Halbordnung 414
Halde 127; 347
Halteprädikat 197
Halteproblem 17
Hamiltonscher Zyklus 269
Hardware 36; 155
Harmonische Reihe 416
Hash-Funktion 343
Hash-Tabelle 344
Hashing 343
Hauptmodul 57
Hauptprogramm 161

Hauptspeicher 34; 361; 397
HB-Baum 340
Heap 127; 374; 402
Heapsort 374
Hintergrundspeicher 34; 397
Höhe eines Baumes 276; 309

I

Imperative Sprachen 235; 237
Implementierung 159; 162
Import 57; 61; 88; 160
Index (eines Feldes) 111
Information hiding 167
Initialisierung 172; 200; 215
Inkarnation 89; 90
innerer Knoten 276
Inorder 279, 303
INTEGER 62; 72
interne Sortiervverfahren 368; 369
internes Sortieren 361
Interpreter 39; 244; 249
Invariante 207; 213
invers geordnet 361
Inversionszahl 363
irreflexiv 413
iterativ 85; 97; 215

K

k-beschränkt 276
k-gleichverzweigt 276
Kante 265
Kantenfolge 267
kantenmarkierter Graph 268
Kantenzug 267
Kapselung 157; 159
kartesisches Produkt 413
Keller 179
Kellerspeicher 82; 127; 174; 402
Keplersche Faßregel 103
Kerninformatik 44
Kette 133
Klasse 157; 236

Knoten 265
 knotenmarkierter Graph 268
 Knotenzahl (eines AVL-Baums) 323
 Knuth-Morris-Pratt (KMP) 292
 Kollision 343; 346; 390
 Kollisionsauflösung 346; 353
 Kollisionsbehandlung 346
 Kommentar 61
 Kommunikationsprotokoll 43
 kompatibel 41; 126
 Komplementierer 32
 komplexe Datentypen 107
 Komplexität 255; 261
 Komplexitätstheorie 256
 Konfiguration (einer Maschine) 197
 Konstante 65
 Kontextbedingung 59
 kontextfreie Grammatik 25; 58
 kontextsensitive Grammatik 25; 59
 Korrektheit 157; 196; 210; 223
 kürzeste Wege 269

L

Laden 161
 längste Wege 269
 Latenzzeit 333
 Lauf 393
 Laufschleife 95; 111
 Laufzeit 87; 259
 Laufzeitfehler 79; 86; 91
 Laufzeitsystem 39; 87
 Lebensdauer 87; 90; 94; 127
 leere Anweisung 205
 Lese-Schreibkopf 13
 Lexikalische Analyse 38
 Lexikographische Ordnung 391
 LIFO-Prinzip 179
 linear-zeitbeschränkt 260
 lineare Ordnung 414
 lineares Sondieren 348
 lineares Suchen 287
 Links-Rechts-Rotation 312
 Links-Rotation 311

Liste 133; 140
 Literal 65
 Logik-basierte Sprachen 236; 243
 lokal 94
 lokaler Speicherbereich 402
 Lokalität 94; 102; 121
 LOOP-Schleife 96

M

Marke 100
 Markierungsmethode 402
 Maschine 9; 11; 194
 Maschinencode 28; 156
 Matching-Problem 269
 Matrix-Multiplikation 114
 Mehrwege-Mischen 395
 Menge 107
 Metasymbol 24; 51
 Methode 157; 246
 Mikroprogrammeinheit 31
 Mischen 362; 393
 Mischphase 394
 Modul 57; 61; 152; 159
 MODULA-2 21; 55; 417
 Modularisierung 159; 161; 165
 Monoid 22

N

Nachbedingung 195; 203; 215
 Nachricht 236
 Natürliches Mischen 395
 Nebenwirkung 72
 NEW 128
 nicht-berechenbare Funktionen 17
 Nichtdeterminismus 251; 260; 262
 nichtdeterministisch 10; 259
 Nichtterminalsymbol 23; 25; 53
 NP-hart 264
 NP-Sprache 263
 NP-vollständig 264; 270

O

obere Schranke 259
Objekt 93; 236; 246
Objektklasse 246
objektorientierte Sprachen 236; 246
opak 173; 174
operationale Semantik 194; 196; 198
Operationsteil 30
optimaler Suchbaum 328
Ordnungskriterium 364
Ordnungsverträglichkeit 363
Organisationsprogramme 37
Orientierung (in Graphen) 265
Overflow-Behandlung 407

P

P-NP-Problem 263
Paradigma 235
Parameter 71; 136; 156; 357
partielle Funktion 13; 73; 76
partielle Korrektheit 210
partielle Ordnung 414
PASCAL 148
Pattern 291
Perfektes Hashing 345
periphere Geräte 36
Permutation 361; 365
Pointer 128
polynomial-zeitberechenbar 261
polynomial-zeitbeschränkt 260
Portabilität 11; 157
Positionssystem 22
Postorder 279
Postorder-Grad-Darstellung 283
Prädikat 195; 236; 243
Praktische Informatik 44
Prämisse (einer Regel) 243
Preorder 279
Preorder-Postorder-Darstellung 285
Preorder-sequentielle-Darstellung 286
Primärindex 345

Primärkollision 351
Primärschlüssel 364
privat 167; 173
Produktion 24; 52
Programm 21; 57
Programmablaufplan 228
Programmierparadigmen 235
Programmiersprachen 11; 19; 235; 252
Programmierstil 94; 121
Programmodul 161
Programmverzweigung 66
Prozedur 57; 61; 72; 102; 156
Prozeduraufruf 61
Prozedurkopf 102
Prozedurtyp 102
Prozedurvariablen 103
Prozessor 31
Pseudocode 78
Puffervariable 148

Q

quadratisches Sondieren 349
Quellprogramm 192
Querübersetzer 38
Quicksort 378

R

REAL 62
Rechner 9
Rechnerarchitektur 254
Rechnernetz 41
Rechnersystem 36
Rechts-Links-Rotation 312
Rechts-Rotation 311
Record 117; 119; 121; 124
Reduzierbarkeit 263
Referenz-Parameter 86; 93; 94; 136
reflexiv 413
regelbasierte Sprachen 236; 248
Regel 236; 243; 248
Regelsystem 250
Regenerationszeit 34

Register 32
 Regressionstest 224
 Rekursion 78; 87; 93; 207
 rekursiv 60; 78; 97; 107
 rekursive Funktion 16
 Relation 413
 REPEAT-Schleife 96
 rotating-first-fit 399
 Rotation 311; 343
 Rucksackproblem 264
 Rundungsfehler 73

S

Sammelphase 389
 Schlange 174
 Schleifenformen 212
 Schleifenkonstruktion 215
 Schleifenrumpf 210
 Schlinge 265
 Schlüssel 287; 302; 332; 343; 364
 Schlüssel-Verweis-Tabelle 365
 Schlüsseltransformation 343
 Schlüsselvergleich 302
 schnelle Sortiervverfahren 373
 Schnittstelle 159
 Schnittstellentest 227; 232
 Schrittzahl 259; 364
 Seite 397; 401
 Seitenaustauschverfahren 397
 Sekundärkollision 351
 Sekundärspeicher 34
 Selektor 117; 125
 semantische Analyse 39
 semantische Funktion 195
 Semantik 19; 191; 203
 Sentinel 288
 separate Übersetzung 159
 Sequenz 206
 Set 107
 Shakersort 372
 Shellsort 373
 Simpson-Integration 103
 Sohn 276

Sondieren 348
 Sondierfolge 348
 Sortieren 361
 Speicher 33
 Speicheradressierung 401
 Speicherbelegung 128
 Speicherbereinigung 398; 402; 406
 Speicherkomplexität 259
 Speicherplatz 89; 337; 344
 Speicherplatzanforderung 397
 Speicherverwaltung 397
 Spezifikation 191; 204; 221; 224
 Sprache 9; 20; 27
 Sprung 30; 100; 230
 Stabilität 363
 Stack 174; 179; 405
 stark zusammenhängend 268
 Startknoten 267
 Startsymbol 25
 Statusvariablen 102
 Stirlingsche Formel 414
 Stream 152
 Streuen und Sammeln 362; 389
 Streuphase 389
 String 56; 113; 291
 Suchen in Tabellen 287
 Suchen von Zeichenketten 291
 Suchpfad 312
 Suchverfahren 287; 291; 302
 Suchzeit 309; 327
 Summenformeln 415
 symbolische Konstante 62; 65
 Symboltabelle 38
 symmetrisch 414
 syntaktische Analyse 38
 Syntax 19; 192
 Syntax von MODULA-2 417
 Syntaxdarstellungen 51
 Syntaxdiagramm 25; 51; 417

T

Tabelle 287
 Technische Informatik 45

Teilen und Herrschen 289
Teilgraph 265
Terminalsymbol 23; 25; 53
Terminierung 10; 210
Test 191; 222; 232
Testdaten 226
Textfile 147
Theoretische Informatik 44
Tiefenbeschränkung 341
totale Korrektheit 210
transitiv 414
transitive Hülle 270
Traversieren 279
Trennphase 394
TRUE 74
Tupel 413
Turing-Maschine 9; 11
Turing-Programm 13
Typ 71
Typbindung 77
Typgleichheit 112
typkompatibel 85
Typtransfer-Funktion 76

U

Überdeckung 227
Überdeckungsgrad 230
Übergangsfunktion 197
Übergangstabelle 13
Überlauf 73; 136; 202; 402
Überlaufbehandlung 334
Übersetzer 28; 38; 192; 204
Übersetzersemantik 194; 196
Übersetzung 159; 161
Übertragungszeit 333
unformatierte Ein-/Ausgabe 148
untere Schranke 259

V

Variablen-Transformator 210
Variable 76; 85
Variablenkonzept 235

variante Records 123
Vater 276
Verbund 117
Vererbung 158
Verifikation 202; 204; 208; 212; 221
Verweiszählermethode 402
Verzweigung (im Programm) 66
Vielweg-Baum 332
virtuelle Maschine 11
virtuelle Speicherverwaltung 397; 401
vollständig ausgeglichen 276; 309
vollständig ausgeglichener Baum 366
vollständiger k-Baum 276
vollständiger Teilgraph 265
von-Neumann-Rechner 29; 155; 235
Vorbedingung 195; 202; 215

W

Vorsortierung 364
Wächter 288
Wald 275
Warteschlange 170
weakest precondition 202
Weg 267
wegäquivalent 270
Wegeüberdeckung 230; 233
Wert-Parameter 86; 94
Wertzuweisung 85; 112; 128; 205
WHILE-Schleife 95
Wiederholbedingung 210; 213; 215
WITH-Anweisung 118
worst case 369
Wort 22
Wurzel 275
Wurzelbalance 341

Z

Zahl 22
Zeichen 12; 21
Zeichenreihe 113; 291
Zeiger 127; 130
Zeigervariable 402

Zeitkomplexität 259
Zielprogramm 192
Ziffer 22
Zugriffsart 34
Zugriffsmodi 148
Zugriffswahrscheinlichkeit 326
Zugriffszeit 33
Zusammenhangskomponente 268
Zusicherung 202
Zweigüberdeckung 231; 233
Zweiphasen-Mischen 394
Zyklus (in Graphen) 267; 403
Zykluszeit 34