Technische Informatik

Schiemangk

MODULA-2

VEB Verlag Technik Berlin

Technische Informatik

Herausgegeben von

Dr.-Ing. Ludwig Claßen

Dr.-Ing. Dieter Nedo

Dr. rer. nat. Gerhard Paulin

MODULA-2

Prof. Dr. sc. nat. Hans Schiemangk

VEB VERLAG TECHNIK BERLIN

Schiemangk, Hans
MODULA-2 / Hans Schiemangk. - 1. Aufl. -
Berlin : Verl. Technik, 1989. - 240 S.:
4 Bilder, 6 Taf.
ISBN 3-341-00857-8

ISBN 3-341-00857-8
ISSN 0863-0860 Technische Informatik

1. Auflage

VT 201 · 3/5885-1 (251)
Printed in the German Democratic Republic
Druck und buchbinderische Verarbeitung:
(140) Druckerei Neues Deutschland, Berlin
Lektor: Jürgen Reichenbach
Einbandgestaltung: Gabriele Schwesinger
LSV 3054
Bestellnummer: 554 269 8
02400

Vorwort

Personalcomputer haben in den letzten Jahren massenhafte Verbreitung gefunden. Ein Ende dieser Entwicklung ist nicht in Sicht.
Auf der einen Seite gibt es verstärkt die Tendenz, vorgefertigte Softwarepakete für Datenbanken, Tabellenkalkulation und Textverarbeitung zu verwenden. Andererseits empfindet eine Vielzahl von Computerbenutzern ein sehr starkes Bedürfnis, dem Rechner den eigenen Willen aufzuzwingen, ihn selbst zu programmieren. Die Mehrheit der für Personalcomputer Programmierenden steigt über BASIC in die Sache ein.

Das vorliegende Buch wendet sich an Leser, die bereits mit einem Computer umgehen können und schon ein wenig in irgendeiner Programmiersprache programmiert haben.
MODULA-2 ist eine kleine, überschaubare Sprache, für die es sehr gute Compiler gibt, die auf Rechnern aller Grössenklassen laufen. MODULA-2 unterstützt den Aufbau von Programmen aus Bausteinen; vielfältige Daten- und Ablaufstrukturen lassen sich elegant formulieren. Die Beschäftigung mit MODULA-2 kann ein Schritt zum Einstieg in die aktuelle Softwaretechnologie sein.
Beim Aufschreiben des Textes hatte ich einen an der Sache interessierten, mitdenkenden Leser vor meinem geistigen Auge, der vieles sofort am Computer ausprobiert. Deshalb habe ich meinen Compiler mit allen abgedruckten Programmen konfrontiert und sie nach bestem Wissen und Gewissen getestet. Ich hoffe, dass sich die Menge der Verärgerungen wegen fehlerhafter Beispielprogramme in Grenzen hält, und bin für jeden einschlägigen Hinweis dankbar. Kontaktadresse: Prof. Hans Schiemangk, Sektion Mathematik, Humboldt-Universität zu Berlin, Postfach 1297, Berlin, DDR-1086.

Die hinter dieser Darstellung der Sprache MODULA-2 stehende Haltung zum Programmieren und zur Softwaretechnologie hat sich bei mir in vielen Gesprächen und durch gemeinsame Arbeit am Bereich Informationsverarbeitung der Sektion Mathematik der Humboldt-Universität zu Berlin herausgebildet. Viele Einsichten verdanke ich meinen Kollegen und Studenten.

Ein besonderes Dankeschön geht an die Adresse des Mitherausgebers der Reihe "Technische Informatik", Herrn Dr. Gerhard Paulin, der durch Diskussionen und manche Mahnung einen grossen Anteil am Entstehen dieses Lehrbuchs hat. Von ihm habe ich während der letzten 25 Jahre nicht nur gehobene Elementargeometrie für Mathematik-Olympiaden oder Maschinenprogrammierung für aus heutiger Sicht exotische Rechner, sondern auch vieles über das Lehren des Programmierens gelernt.
Ich danke dem Verlag Technik, insbesondere Herrn Jürgen Reichenbach und Herrn Hartmut Heinrich, für die verständnisvolle und gedeihliche Zusammenarbeit auf dem langen Weg bis zum Manuskript und zum fertigen Buch.

Berlin, Dezember 1988 Hans Schiemangk

Inhaltsverzeichnis

1. Einführung

Die Programmiersprache MODULA-2 gefällt mir. Sie ermöglicht den Aufbau von Programmen aus einzeln übersetzten Programmbausteinen. Die Verträglichkeitsprüfungen durch den Compiler beschränken sich nicht auf den jeweiligen Baustein; sie erfolgen auch über Bausteingrenzen hinweg. Ausserdem hat MODULA-2 die Stärken der Sprache PASCAL geerbt: PASCAL ist die Oma von MODULA-2.

1.1. Grobcharakteristik von MODULA-2

Die bekannten technologischen Fortschritte bei Entwurf und Produktion von Baugruppen und Geräten zur Informationsverarbeitung sowie zur Informationsgewinnung , -übertragung und -speicherung tragen wesentlich zum lawinenhaften Ansteigen des Bedarfs an Programmen bei. Nach wie vor haben wir weltweit ein Softwareproblem. Das Ende dieser Entwicklung ist nicht absehbar. Ein anerkannt wichtiger Aspekt bei der Lösung des Softwareproblems betrifft die Fragen nach Portabilität und Mehrfachnutzbarkeit von Programmen:

1. Portabilität
   Wie kann erreicht werden, dass ein Programm unter Beibehaltung seiner Funktionseigenschaften ohne Umprogrammierung in eine andere Hardware/Software-Umgebung übertragbar ist?
2. Mehrfachnutzbarkeit
   Wie kann erreicht werden, dass funktionsgleiche Programmteile ohne Neuprogrammierung in unterschiedlichen Programmen verwendbar sind?

Portabilität würde den Nutzern von Rechentechnik das Leben mit vielfältiger Hardware und unterschiedlichen Betriebssystemen erträglicher machen. Hier sind die Compileranbieter gefordert! Einige Firmen vertreiben quelltextkompatible MODULA-2-Systeme, mit denen ein gewisses Spektrum von Rechnern und Betriebssystemen überstrichen wird (z. B. IBM/PC- und VAX/11-kompatible Computer sowie MS/DOS-, UNIX- und VMS-kompatible Betriebssysteme).
Wie sieht es bez. der mehrfachen Verwendung von Programmteilen aus? Die Definition von MODULA-2 enthält programmiersprachliche

Ausdrucksmittel, die es jedem Programmierer nahelegen, Programme aus selbstgeschriebenen oder anderweitig verfügbaren Bausteinen zusammenzusetzen. Jeder Baustein kann in unterschiedlichen Programmen verwendet werden.
Für "Baustein" ist auch die Bezeichnung "Modul" sehr gebräuchlich; daher der Name "MODULA-2". Das Zerlegen von Programmen in Bausteine nennt man "Modularisierung". In diesem Sinne sind auch die Schlagwörter "modulares Programm" und "modulare Programmierung" zu verstehen.
Das Bausteinkonzept von MODULA-2 leistet mehr als die bei FORTRAN und PASCAL bekannten Varianten der separaten Compilation von Unterprogrammen. In beiden Sprachen kann der Compiler z. B. nicht feststellen, ob die Anzahl der im Aufruf eines separat übersetzten Unterprogramms angegebenen Parameter "richtig" ist. Es könnte sein, dass dieses Unterprogramm in Wirklichkeit eine andere Parameteranzahl verlangt. Jeder MODULA-2-Compiler führt derartige die Grenzen von Bausteinen überschreitende Verträglichkeitsprüfungen aus! Wie funktioniert das? Für jeden Baustein wird zuerst ein sogenannter Definitionsteil geschrieben. Er enthält alle Angaben, die der Compiler für Verträglichkeitsprüfungen des oben angegebenen Typs benötigt. Der Compiler überprüft diesen Definitionsteil und legt ihn codiert im Filespeicher ab. In zwei Situationen greift der Compiler automatisch auf die dem Definitionsteil eines Bausteins entnommene Information zurück:

1. wenn der zu dem Baustein gehörende sogenannte Implementationsteil übersetzt wird
2. wenn bei der Übersetzung eines Programms oder eines anderen Bausteins Bezugnahmen auf den fraglichen Baustein entdeckt werden.

Der Definitionsteil schreibt die Schnittstelle zwischen den Nutzern eines Bausteins (Hauptprogramme sowie andere Bausteine) und der Realisierung des Bausteins (Implementationsteil) fest. Der Compiler sichert, dass beide Seiten sich an diesen "Vertrag" halten. Hieraus ergibt sich ein wesentlicher softwaretechnologischer Vorzug der Sprache MODULA-2. Sie erleichtert die Organisation der Arbeit von Programmiererteams. Grob gesagt, sieht das so aus:

> Der Chef macht sich Gedanken über die Zerlegung des Programmierauftrags in einzelne Bausteine und schreibt die zugehöri-

gen Definitionsteile. Diese werden übersetzt und bilden nun die rechnerintern vorliegende verbindliche Grundlage für die Arbeit aller Programmierer. Einige haben Programme zu schreiben, in denen die vorgegebenen Bausteine benutzt werden. Andere müssen Bausteine realisieren. Die Prüfung, ob jedermann sich an die vom Chef vorgegebenen Schnittstellen hält, obliegt dem Compiler, und an dem kann keiner vorbei.

Wenn ein Softwareteam längere Zeit mit MODULA-2 arbeitet, dann entstehen Bausteinbibliotheken, auf die häufig zurückgegriffen wird. Bei der Pflege solcher Bibliotheken fällt ein weiterer Vorzug des Bausteinkonzepts von MODULA-2 ins Gewicht:

Fehlerkorrekturen und Verfahrensverbesserungen im Implementationsteil eines Bausteins erfordern keine Neuübersetzung der "Bausteinbenutzer", solange der Definitionsteil unangetastet bleibt.

Auch diese Eigenschaft ist günstig für die Teamarbeit. Ein neukonzipierter Baustein erhält zunächst einen "einfachen" Implementationsteil. (In letzter Zeit wird so etwas oft "Prototyp" genannt.) Auf dieser Basis sind schon bald Erprobungsläufe für all jene Programme und Bausteine möglich, die den fraglichen Baustein benutzen. Parallel dazu kann an ausgefeilten Realisierungen des Bausteins gearbeitet werden. Sie finden später an Stelle der Prototypen Verwendung in den "heissen" Programmsystemen.

Bausteinkonzepte betreffen das "Programmieren im Grossen". Für das "Programmieren im Kleinen" (welche Daten- und Ablaufstrukturen sind zur Realisierung der Bausteine verfügbar?) bietet MODULA-2 etwa dasselbe wie PASCAL. Miteinander beliebig kombinierbare Record- und Arraystrukturen sowie Pointer gestatten die Konstruktion vielfältiger statischer und dynamischer Daten. Auswahlanweisungen, bedingte Anweisungen und vier Sorten von Zyklusanweisungen sowie das Prozedurkonzept mit Wert- und Referenzparametern erlauben die überschaubare Notation von Programmabläufen.

Eine explizite Sprunganweisung gibt es nicht; RETURN- und EXIT-Anweisungen zum Verlassen von Prozeduren und bestimmten Zyklen sind ein ausreichender Ersatz. Eingabe/Ausgabe-Operationen werden sämtlich durch spezialisierte Bausteine abgewickelt. Files und Filetypen gehören nicht zur Sprache MODULA-2.

Die folgende informale "Gleichung" versucht, alle zwischen den Programmiersprachen MODULA-2 und PASCAL bestehenden Unterschiede und Gemeinsamkeiten zusammenzufassen:

MODULA-2 = PASCAL
- Files u. E/A-Anweisungen
- Marken u. GOTO
+ Bausteinkonzept
+ Prozedurtypen u. -variablen
+ Coroutinen
+ LOOP-, EXIT- u. RETURN-Anweisung
+ Konstantenausdrücke
+ lokale Module
+ maschinennahe Ausdrucksmittel
+ "syntaktischer Zucker".

Prozedurtypen und -variablen stellen eine Verallgemeinerung der schon seit ALGOL-60 bekannten formalen Prozeduren dar. Sie können in bisher ungewohnter Weise zur Parametrisierung von Programmteilen benutzt werden. Ich habe die Erfahrung gemacht, dass die Flexibilität mancher Bausteine durch die Verwendung geeigneter Prozedurvariablen erstaunlich vergrössert wird.
Die sehr elementaren Möglichkeiten zur Arbeit mit Coroutinen in MODULA-2 sind als Grundelemente für die Realisierung von Prozess- und Synchronisierungskonzepten gedacht. Sie stellen ein i. allg. selten genutztes Mittel zur Strukturierung von Programmabläufen dar. Der letzte "Summand" meint, dass sich einige Dinge in MODULA-2 etwas anders schreiben als in PASCAL, obwohl dasselbe ausgedrückt wird. Ich empfinde diese Syntaxveränderungen als Fortschritt.

1.2. Fahrplan des Buches

Abschnitt 2 enthält ein sehr ausführlich erläutertes Programmbeispiel mit zwei Teilen:

1. Programmbaustein für die Dialogeingabe von Zahlen und Zeichenketten sowie
2. Hauptprogramm zur Eingabe und Abspeicherung von Städtenamen und Einwohnerzahlen.

Das Hauptprogramm nutzt die Dienste des Bausteins.

Abschnitt 3 wendet sich vor allem an jene Leser, die über einen Computer mit MODULA-2-System verfügen und die Lektüre des Buches mit praktischen Experimenten begleiten wollen. Ich empfehle, die erläuterten Schritte bis zur Abarbeitung eines Programms nachzuvollziehen. Gönnen Sie sich erst dann ein wenig Ruhe, wenn nach dem Start des Programms die Zeile

Hier bin ich!

auf Ihrem Bildschirm erschienen ist und Sie den gesamten Vorgang vom Quelltext bis zur erfolgreichen Abarbeitung dreimal wiederholt haben! Wer sich nur theoretisch über MODULA-2 informieren will, der kann diesen Abschnitt überblättern.

Abschnitt 4 enthält eine Einführung des "Erweiterten Backus-Naur-Formalismus" (EBNF) zur Syntaxdarstellung und die Definition der Lexik von MODULA-2.

Nach all diesen notwendigen Präliminarien folgt die systematische Sprachdarstellung von MODULA-2. Hauptgegenstand des fünften Abschnitts sind Variablen-, Konstanten- und Unterprogrammvereinbarungen. Besonders genau sollte man die Frage der Gültigkeitsbereiche von Bezeichnern (Abschn. 5.4) studieren.

Im Abschnitt 6 folgt die Erläuterung der Standarddatentypen von MODULA-2. Vieles ist ähnlich wie in anderen Programmiersprachen; aber den Prozedurtyp 'PROC' (Abschn. 6.6) und das zugehörige Beispiel sollten Sie sich genauer ansehen.

Abschnitt 7 will neben der Darstellung von Ausdrücken - das sind formelmässige Berechnungsvorschriften - auch Unterprogramme, die sich selbst aufrufen, ins rechte Licht setzen und ihnen den Hauch des Exotischen nehmen.

Als Höhepunkt des Abschnitts 8 "Anweisungen" stelle ich eine Form der syntaxorientierten Programmierung vor: Aus Datenbeschreibungen in EBNF können systematisch Verarbeitungsprogramme abgeleitet werden. Der dieser Problematik gewidmete Abschnitt 8.4 kann beim ersten Lesen übergangen werden.

Die Ausdrucksfähigkeit einer Programmiersprache hängt wesentlich von den Möglichkeiten zur Beschreibung von Datenstrukturen ab. Abschnitt 9 befasst sich mit den Datentypen von MODULA-2. Als besonders wichtig möchte ich die Abschnitte 9.4 und 9.5 über Prozedurtypen und Bedingungen für die Verträglichkeit von Typen hervorheben.

Die umfassende Darstellung des Bausteinkonzepts von MODULA-2 ist Gegenstand des Abschnitts 10. Besondere Aufmerksamkeit verdient Abschnitt 10.2. Ich erläutere den Datenfluss bei der Compilation: Welche Files liest der Compiler, welche Files erzeugt er? Es geht hierbei um die Prinzipien, nach denen die Bausteinphilosophie von MODULA-2 funktioniert. Ein umfangreiches Beispiel (Sortieren in Arrays und Zeitmessungen zum Vergleich von Sortierverfahren) demonstriert die Arbeit mit Programmbausteinen.
Abschnitt 11 befasst sich mit Grundproblemen des numerischen Rechnens. Am Anfang stehen die rechnerinterne Darstellung von REAL-Zahlen und damit verbundene missliche Effekte. Es folgt ein vollständig ausprogrammierter Baustein zur Zahlkonvertierung.
Die Behandlung von Zeiger- oder Pointertypen zum Aufbau dynamischer Datenstrukturen im Abschnitt 12 ist recht ausführlich und enthält viele einfache Beispiele. Lehrerfahrungen mit PASCAL und MODULA-2 besagen, dass Lernende hier grössere Verständnisschwierigkeiten haben als bei einem so einfachen Konzept zu erwarten wäre. Am Schluss dieses Abschnitts und damit auch am Schluss der Stoffvermittlung in diesem Buch werden binäre Bäume als Beispiel für eine dynamische Datenstruktur vorgestellt.
Sehr stiefmütterlich behandle ich lokale Module und maschinennahe Ausdrucksmittel. Beides wird im Abschnitt 13 nur der Vollständigkeit halber angesprochen. Praktische Erfahrungen zeigten, dass beide Dinge zu schwer durchschaubaren Programmen verleiten. Die Verwendung maschinenbezogener Sprachelemente steht ausserdem in direktem Widerspruch zum Wunsch nach Portabilität.
Eine ausführliche Behandlung des Coroutinenkonzepts von MODULA-2 ist den Randbedingungen für den Umfang dieses Bandes zum Opfer gefallen. Auch Coroutinen sind nichts Exotisches. Ihre Verwendung als Mittel zur Strukturierung des Programmablaufs kann wesentlich zu überschaubareren Programmen beitragen. Da zum glaubhaften Vermitteln dieser Einsichten ein paar umfangreichere Beispielprogramme diskutiert werden müssten und eine "geraffte Darstellung" nicht über das hinausgehen würde, was ohnehin in der Sprachbeschreibung steht, habe ich mich entschlossen, ganz auf das Coroutinenkonzept zu verzichten. Eine tiefgründige Diskussion der MODULA-2-Coroutinen und die Beschreibung der Implementation mehrerer Prozesskonzepte findet man in der Dissertation [Ahre85].

Apropos: Sprachbeschreibung. Der "Urtext" [Wirt80] mit den Ergänzungen und Veränderungen [Wirt84] ist bis zum erfolgreichen Abschluss der gegenwärtigen internationalen Standardisierungsbestrebungen die verbindliche Quelle. Als sehr gelungene deutschsprachige Version einer MODULA-2-Sprachbeschreibung empfinde und empfehle ich die Schrift [Robo87a].

2. Ausführlich kommentiertes Programmbeispiel

Ausgehend von der Aufgabenstellung des Beispielprogramms sowie einem Dialogprotokoll und dem zugehörigen Ergebnisfile werden die Quelltexte aller Bestandteile des Programms vollständig angegeben und erläutert. Der Standardbaustein 'InOut' für Eingabe- und Ausgabeoperationen mit Terminal und Textfiles (Abschn. 2.4) wird auch in den weiteren Kapiteln oft benutzt.
Praktische Hinweise zur Inbetriebnahme des Beispielprogramms und einige Testläufe sind Gegenstand der Abschnitte 3.2 und 3.3.

2.1. Aufgabenstellung, Dialogprotokoll und Ergebnisfile

Verlangt sei ein Programm zur Erfassung von Städtenamen und Einwohnerzahlen. Der an Tastatur und Bildschirm arbeitende Nutzer des Programms soll im Dialog mit dem Rechner zuerst nach dem Namen einer Stadt und dann nach der Einwohnerzahl gefragt werden. Dieser Vorgang ist zu wiederholen, bis als Stadtname ein speziell vereinbartes Endekennzeichen eingegeben wird. Anschliessend sind die erfassten Wertepaare

Einwohnerzahl Stadt

in Tabellenform der Grösse nach absteigend sortiert auf ein File mit vorgegebenem Namen zu schreiben. Kleinere Dienstprogramme dieser Art werden bei der Arbeit an Personalcomputern immer wieder einmal benötigt. Die folgenden Zeilen enthalten das Bildschirmprotokoll eines Dialogs mit dem Programm zur Erfassung von Städten und Einwohnerzahlen.

```
Stadt : Elsterwerda
Einwohnerzahl : 10500
Stadt : Wahrenbrueck
Einwohnerzahl : 650
Stadt : Doberlug-Kirchhain
Einwohnerzahl : 0
Einwohnerzahl : 9200
Stadt : #
```

Der erste Teil jeder Zeile bis zu dem Leerzeichen hinter dem Doppelpunkt ist die "Frage" des Programms. Danach folgt jeweils die Nutzerantwort. Offensichtlich unsinnige Einwohnerzahlen werden nicht akzeptiert: Wiederholung derselben Frage. Als Endekennzeichen wurde ein Doppelkreuz (#) auf der Position des ersten Buchstaben eines Stadtnamens festgelegt.
Der Inhalt des hierbei geschriebenen Files besteht aus den folgenden drei Zeilen.

```
10500  Elsterwerda
 9200  Doberlug-Kirchhain
  650  Wahrenbrueck
```

Wie kann eine MODULA-2-Lösung dieser Programmieraufgabe aussehen?

2.2. Definition eines Bausteins

Betrachten wir noch einmal die Aufgabenstellung für unser Programm. Zwei Teilaufgaben, die sicherlich auch in ganz anderen Zusammenhängen auftreten können, sind erkennbar:

1. Erfassen eines Namens und
2. Erfassen einer ganzen Zahl.

Diese Einsicht ist Anlass für den Entwurf und die Programmierung eines Bausteins 'Fragen'. Programm 2.1 zeigt den Definitionsteil dieses Bausteins. Der Definitionsteil eines Bausteins heisst in der MODULA-2-Terminologie "Definitionsmodul".
Es folgen ausführliche Erläuterungen zum Programm 2.1:

1. Die Zeilennummern (1 bis 23) gehören nicht zum Programm! Sie wurden als Hilfsmittel für Erläuterungen hinzugefügt. Das trifft auch auf alle weiteren Programme mit Zeilennummern zu.
2. Zeichenfolgen, die mit (* beginnen und mit *) enden, sind Kommentare. Wenn man sie wegläßt, ändert das nichts an der Bedeutung des Programms.
   Die Zeilen 2 bis 5, 7 bis 15 und 22 sind Kommentare. Programme sollten zur Verbesserung der Lesbarkeit Kommentare enthalten. Auch Einrückungen wie in den Zeilen 17, 20 und 21 sowie eingestreute Leerzeilen (Zeile 18) können ein Programm übersichtlicher machen.

```
 1 DEFINITION MODULE Fragen;
 2 (*-------------------------------------------------------------
 3    Prozeduren zur Dialogeingabe von Zeichenketten und ganzen
 4    Zahlen
 5  -------------------------------------------------------------*)
 6 EXPORT QUALIFIED TextFrage, ZahlFrage;
 7 (*-------------------------------------------------------------
 8    Alle Prozeduren arbeiten nach folgendem Ablaufschema:
 9      1. Frage   - Anforderungstext auf das Terminal schreiben
10      2. Antwort - entsprechende Einheit lesen
11      3. Pruefen - Zahlen bez. unterer u. oberer Grenze testen
12    Fehlerhafte Eingaben fuehren zur Wiederholung des Ablaufs;
13    die Prozeduren werden erst dann verlasssen, wenn akzeptable
14    Werte eingegeben wurden.
15 -------------------------------------------------------------*)
16 PROCEDURE TextFrage(anforderung: ARRAY OF CHAR;
17                     VAR text: ARRAY OF CHAR);
18
19 PROCEDURE ZahlFrage(anforderung: ARRAY OF CHAR;
20                     untereGrenze,obereGrenze: INTEGER;
21                     VAR zahl: INTEGER);
22 (*-------------------------------------------------------------*)
23 END Fragen.
```

Programm 2.1. Definitionsteil des Bausteins 'Fragen'

3. Ein Definitionsmodul beginnt mit einem Kopf (Zeile 1), der aus den Schlüsselwörtern DEFINITION und MODULE, dem Bausteinnamen sowie einem Semikolon besteht. Am Modulende (Zeile 23) stehen das Schlüsselwort END, der Bausteinname und der abschliessende Punkt.
4. In Zeile 6 sind hinter den Schlüsselwörtern EXPORT und QUALIFIED alle jene Bezeichner aufgezählt, die vom Baustein 'Fragen' exportiert, d. h. zur Benutzung durch andere Bausteine oder Hauptprogramme zur Verfügung gestellt werden. In unserem Fall sind das 'TextFrage' und 'ZahlFrage'. Trennzeichen innerhalb der Aufzählung ist das Komma. Am Schluss steht ein Semi-

kolon. Es gibt Versionen von MODULA-2-Compilern, bei denen die EXPORT-Klausel nicht obligatorisch bzw. ganz verboten ist: Alle im Definitionsmodul auftretenden Bezeichner werden exportiert! Das ist eine vernünftige Regelung.

5. In den Zeilen 16 und 17 sowie 19 bis 21 ist für alle Programmierer, die auf Leistungen des Bausteins 'Fragen' zurückgreifen wollen, und für den MODULA-2-Compiler aufgeschrieben, worum es sich bei den Bezeichnern 'TextFrage' und 'ZahlFrage' handelt. Exakt ausgedrückt: In den Zeilen 16 und 17 ist genau die Information enthalten, die der Compiler für die Entscheidung benötigt, ob 'TextFrage' in einem anderen Baustein bzw. Hauptprogramm in zulässiger Weise verwendet wird oder nicht.
6. 'TextFrage' ist der Name einer Prozedur, d. h. eines Unterprogramms (Schlüsselwort PROCEDURE). Hinter dem Prozedurnamen steht in runden Klammern eine Parameterliste. Parameter dienen dem Datenaustausch mit dem Unterprogramm. 'TextFrage' hat zwei Parameter: 'anforderung' und 'text'. Jede Parameterangabe besteht aus Parametername, Doppelpunkt und Parametertyp.
7. Der Begriff "Typ" sollte als "Beschreibung einer Menge von Werten" oder als "Menge von Werten" oder auch als "Wertebereich" gedeutet werden. Der Parametertyp legt fest, welche Werte der Parameter haben darf. 'anforderung' und 'text' können mit Werten aus der durch

   ARRAY OF CHAR

   beschriebenen Wertemenge belegt werden. Es handelt sich um eindimensionale Felder (Schlüsselwort ARRAY), deren Elemente einzelne Zeichen sind (Schlüsselwort OF gefolgt von dem Typnamen 'CHAR'), also um Zeichenketten. 'CHAR' - ein sogenannter Standardtyp - ist Bestandteil der Sprache MODULA-2. Der Wertebereich 'CHAR' besteht aus den auf dem jeweiligen Rechner verfügbaren Zeichen.
8. Die genaue Länge der als Parameterwerte zugelassenen Zeichenketten wird in den Zeilen 16 und 17 nicht festgelegt, sie ist "offen" (keine Angabe eines Indextyps zwischen den Schlüsselwörtern ARRAY und OF), d. h., das Unterprogramm 'TextFrage' kann mit Parametern unterschiedlicher Länge arbeiten.
9. Das Schlüsselwort VAR (Zeile 17) vor einer Parameterbeschreibung legt fest, dass dieser Parameter den Datenaustausch zwi-

schen Prozedur und Umgebung in beiden Richtungen zulässt. Man spricht von einem "Referenzparameter". Fehlt VAR vor einer Parameterangabe, dann handelt es sich um einen sogenannten "Wertparameter". Ein Wertparameter ist ein reiner Eingabeparameter für die Prozedur. Resultate können über ihn nicht nach "aussen" vermittelt werden. 'anforderung' (Zeile 16) ist ein Wertparameter. 'text' (Zeile 17) ist ein Referenzparameter.

10. Die gesamte Konstruktion vom Schlüsselwort PROCEDURE bis zum Semikolon hinter der zugehörigen Parameterliste (Zeilen 16 und 17 bzw. 19 bis 21) wird als "Prozedurkopf" bezeichnet.
11. Der im Prozedurkopf 'ZahlFrage' (Zeilen 19 bis 21) zweimal auftretende Standardtyp 'INTEGER' besteht aus einem rechnerabhängig festgelegten Intervall der negativen und positiven ganzen Zahlen. In Zeile 20 sind zwei Parameter desselben Typs zusammengefasst. Die im Kommentar erwähnte Prüfung (Zeile 11) basiert auf den beiden Wertparametern 'untereGrenze' und 'obereGrenze'. Der Referenzparameter 'zahl' zur Vermittlung des Resultats hat natürlich den Typ 'INTEGER'.
12. Die in einem Programm verwendeten Bezeichner (bisher traten sie als Baustein-, Prozedur- und Parameternamen auf) werden vom Programmierer frei gewählt. Er sollte die Wahl so treffen, dass die Bezeichner etwas über die beabsichtigte inhaltliche Bedeutung der bezeichneten Dinge aussagen.

Die Namenswahl ist nicht ganz beliebig. Schlüsselwörter (MODULE, END, ...) dürfen nicht verwendet werden. Standardnamen (INTEGER, CHAR, ...) sollte man vermeiden. Jeder Programmierer bemerkt sehr bald, dass er einen persönlichen Kompromiss zwischen den Vorteilen "sprechender" Bezeichner und der bei der Programmeingabe auffälligen Unbequemlichkeit langer Bezeichner finden muss.

2.3. Das Hauptprogramm 'Staedte'

Das eigentliche Programm zur Lösung der im Abschnitt 2.1 formulierten Programmieraufgabe greift bez. der Datenerfassung auf die Dienste des soeben fixierten Bausteins zurück. Die eingelesenen Werte werden in einer internen Datenstruktur abgelegt und nach Eingabeende in der gewünschten Reihenfolge ausgegeben. Die Opera-

tionen zum Schreiben des geforderten Ausgabefiles findet man in der zu jedem MODULA-2-System gehörenden Bibliothek von "Standardbausteinen". Programm 2.2 zeigt den Programmodul 'Staedte'. Ein Programmodul ist ein MODULA-2-Hauptprogramm.

```
 1 MODULE Staedte;
 2 (*-------------------------------------------------------------
 3   Namen und Einwohnerzahlen kleiner Staedte im Dialog
 4   eingeben, und diese Angaben der Groesse nach sortiert
 5   auf das File "wohner.tab" schreiben
 6 -------------------------------------------------------------*)
 7 FROM Fragen IMPORT
 8   TextFrage, ZahlFrage;
 9 FROM InOut IMPORT
10   OpenOutput, CloseOutput, WriteInt, WriteString, WriteLn;
11 (*-------------------------------------------------------------*)
12 TYPE
13   Eintrag = RECORD
14               stadt: ARRAY[0..31]OF CHAR;
15               leute: INTEGER;
16             END;
17 CONST
18   Kapazitaet = 70; (* So viele Staedte sind erfassbar *)
19 VAR
20   tabelle: ARRAY[0..Kapazitaet-1]OF Eintrag;
21   position, schluss, maxNr, maxLeute, i: INTEGER;
22 (*-------------------------------------------------------------*)
23 BEGIN
24   position:=0;
25   LOOP (* Dialogeingabe *)
26     WITH tabelle[position] DO
27       TextFrage("   Stadt", stadt);
28       IF stadt[0]="#" THEN EXIT END;
29       ZahlFrage("   Einwohnerzahl", 1, 30000, leute);
30     END (* WITH tabelle... *);
31     position:=position+1;
32     IF position>=Kapazitaet THEN EXIT END;
33   END (* LOOP: Dialogeingabe *);
```

```
34 (* Sortierte Ausgabe *)
35   OpenOutput("wohner.tab");
36   FOR schluss:=position-1 TO 0 BY -1 DO
37     maxNr:=0; maxLeute:=tabelle[0].leute;
38     FOR i:=1 TO schluss DO
39       WITH tabelle[i] DO
40         IF leute>maxLeute THEN maxNr:=i; maxLeute:=leute END;
41       END (* WITH tabelle... *);
42     END (* FOR i     *);
43     WriteInt(maxLeute,8); WriteString("  ");
44     WriteString(tabelle[maxNr].stadt); WriteLn;
45     tabelle[maxNr]:=tabelle[schluss];
46   END (* FOR schluss     *);
47   CloseOutput;
48 END Staedte.
```

Programm 2.2. Programmodul 'Staedte'

Einige Erläuterungen zum Hauptprogramm 'Staedte':

1. Der Kopf eines Programmoduls besteht aus dem Schlüsselwort MODULE, dem Programmnamen und einem Semikolon (Zeile 1). Auch hier stehen am Modulende das Schlüsselwort END, der Modulname und der abschliessende Punkt (Zeile 48).
2. In den Zeilen 7 bis 10 ist notiert, welche Bezeichner woher importiert werden, d. h., welche Dienste welcher Bausteine in Anspruch genommen werden. Dem Schlüsselwort FROM ("von", engl.) folgen der betreffende Bausteinname, das Schlüsselwort IMPORT und die mit einem Semikolon beendete Liste der importierten Bezeichner. Als Trennzeichen innerhalb der Importliste müssen Kommas geschrieben werden. Zur inhaltlichen Bedeutung der Importe vom Baustein 'Fragen' s. Programm 2.1. Die Diskussion der vom Baustein 'InOut' importierten Dinge schiebe ich noch ein wenig auf (s. Punkt 13 und Programm 2.3).
3. Das Schlüsselwort VAR in Zeile 19 leitet Variablenvereinbarungen ein. In Zeile 20 wird eine Variable vereinbart. Ihr Name ist 'tabelle'. Hinter dem Doppelpunkt steht, dass es sich um ein eindimensionales Feld mit 70 Elementen (Index von 0 bis

69) handelt:

```
ARRAY[0..Kapazitaet-1]OF
```

Die Feldelemente sind vom Typ 'Eintrag'. Die obere Indexgrenze 69 ergibt sich aus der Berechnungsvorschrift

```
Kapazitaet-1
```

durch Rückgriff auf die mit dem Schlüsselwort CONST (Zeile 17) eingeleitete Konstantenvereinbarung in Zeile 18. Sie sagt aus, dass im weiteren Programmtext der Konstantenname 'Kapazitaet' geschrieben werden kann und dass sich dahinter der Wert 70 verbirgt. Die Einführung benannter Konstanten trägt zur Verbesserung der Lesbarkeit und der Änderungsfreundlichkeit von Programmen bei.

4. Das Schlüsselwort TYPE (Zeile 12) zeigt an, dass Typvereinbarungen folgen. In Zeile 13 wird der Bezeichner 'Eintrag' vereinbart. Hinter dem Gleichheitszeichen folgt die genaue Beschreibung des Wertebereichs, auf den im weiteren Programm mittels des Bezeichners 'Eintrag' Bezug genommen werden kann. Ein 'Eintrag' ist ein Datensatz (Schlüsselwort RECORD) mit zwei sogenannten Datenfeldern. Das Datenfeld 'stadt' kann Zeichenketten der Laenge 32 als Werte haben (Zeile 14). Das Datenfeld 'leute' kann Werte des Standardtyps 'INTEGER' annehmen (Zeile 15). Die Datensatzbeschreibung wird mit dem Schlüsselwort END und einem Semikolon (Zeile 16) abgeschlossen.
5. In Zeile 21 werden fünf INTEGER-Variablen vereinbart. Bild 2.1 auf der folgenden Seite zeigt die Variablen des Programmoduls 'Staedte' und die in den Zeilen 26, 28 sowie 37 notierten Feldzugriffe.
6. Das Schlüsselwort BEGIN in Zeile 23 markiert den Anfang vom Aktions- oder Anweisungsteil des Programmoduls 'Staedte'. Hier wird festgelegt, wie das Programm arbeiten soll. Die einzelnen Arbeitsschritte sind in Form sogenannter Anweisungen notiert, die durch Semikolons getrennt werden.
7. Die wichtigste Sorte von Anweisungen sind die Ergibtanweisungen, die oft auch "Wertzuweisungen" oder einfach "Zuweisungen" genannt werden. Man erkennt sie am Wertzuweisungsoperator := (Doppelpunkt und Gleichheitszeichen ohne etwas dazwischen), der als "ergibt sich aus" zu lesen ist.

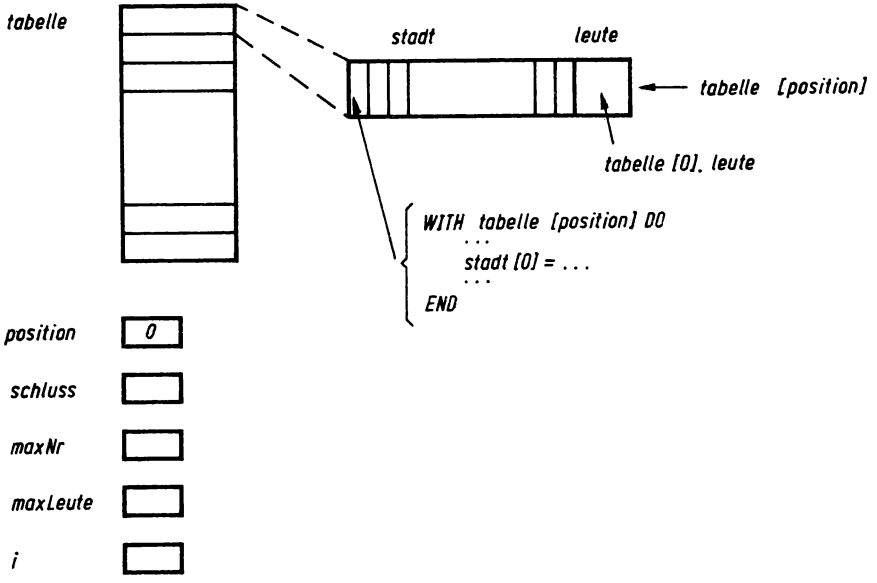


Bild 2.1. Die Variablen des Programmoduls 'Staedte'

Zeile 24 besagt, dass der Variablen 'position' der Wert 0 zugewiesen wird. Oder anders ausgedrückt: Unter dem Namen 'position' wird eine Null abgespeichert.
Sehr interessant ist die Zuweisung in Zeile 31. Sie bewirkt die Erhöhung des Wertes der Variablen 'position' um eins. Die auf der rechten Seite des Wertzuweisungsoperators stehende Berechnungsvorschrift

```
position+1
```

greift auf den z. Z. unter dem Namen 'position' gespeicherten Wert zu und addiert eine Eins. Der so berechnete Wert wird nun wieder unter dem Namen 'position' abgespeichert, also zum neuen Wert der Variablen 'position' gemacht.
Jene Leser, die bereits über Programmiererfahrung verfügen, mögen mir bitte die ausführliche Erörterung von

```
x := x+1
```

verzeihen. Ich habe beim Lehren des Programmierens die Erfah-

rung gemacht, dass das Verstehen des durch diese Anweisung ausgedrückten Sachverhalts die entscheidende Hürde auf dem Wege zum algorithmischen Denken ist. Jedermann hat schon als Schulkind im Mathematikunterricht gelernt, dass für Zahlen die Gleichung

```
x = x+1
```

einen Widerspruch darstellt. Und plötzlich soll beim Programmieren am Computer

```
x := x+1
```

etwas Sinnvolles sein? Der Witz steckt in einem kleinen Unterschied: Die Gleichung will einen Zustand beschreiben; die Ergibtanweisung hingegen beschreibt einen Vorgang.

8. Der Aktionsteil des Moduls 'Staedte' besteht im wesentlichen aus zwei Programmschleifen. Die erste (Zeilen 25 bis 33) realisiert die Dialogeingabe der Namen und Einwohnerzahlen sowie deren Abspeicherung in der Tabelle. Die zweite Schleife (Zeilen 36 bis 46) befasst sich mit der sortierten Ausgabe auf ein File. In den Zeilen 24 und 35 werden notwendige Startbedingungen für den jeweiligen Zyklus geschaffen.
   Das Schlüsselwort LOOP in Zeile 25 leitet einen Zyklus ein, der bis zum zugehörigen END (Zeile 33) reicht. LOOP-END kann mit einem Motor verglichen werden: Die Anweisungsfolge zwischen LOOP und END wird immer wieder abgearbeitet. Ein Ausstieg aus diesem Karussell ist nur mit einer EXIT-Anweisung möglich. Diese Anweisung tritt in zwei unterschiedlichen Situationen auf: Schlüsselwort EXIT in den Zeilen 28 und 32.
9. Die Variable 'position' enthält stets die Nummer des ersten noch nicht vollständig belegten Tabelleneintrags. Der Anfangswert fuer 'position' ist 0 (Zeile 24). Die in den Zeilen 26 bis 30 notierte WITH-Anweisung "füllt" den zur Nummer 'position' gehörenden Tabelleneintrag (s. Punkt 10). In Zeile 31 wird 'position' um eins weitergezählt.
   Die Schleife zur Dialogeingabe muss spätestens dann verlassen werden, wenn die Tabelle voll ist. Dem trägt die IF-THEN-END-Konstruktion ("IF-Anweisung" oder auch "bedingte Anweisung") in Zeile 32 Rechnung: Falls gilt (Schlüsselwort IF), dass

   ```
   position>=Kapazitaet
   ```

   ist, dann (Schlüsselwort THEN) wird die EXIT-Anweisung

(Schlüsselwort EXIT) abgearbeitet. Sie bewirkt, dass hinter dem in Zeile 33 stehenden Schleifen-END weitergemacht wird. Andernfalls wird hinter dem die IF-Anweisung abschliessenden END fortgesetzt. In diesem Fall ist aber das Schleifen-END erreicht, und die Abarbeitung beginnt wieder am Schleifenanfang (Zeile 26).

10. Die WITH-DO-END-Konstruktion gestattet in den zwischen DO und END notierten Anweisungen eine direkte Bezugnahme auf die Datenfelder (vgl. Punkt 4) des zwischen WITH und DO angegebenen Datensatzes. Die Angabe

```
tabelle[position]
```

in Zeile 26 wählt das zur Nummer 'position' gehörende Element des Feldes 'tabelle' aus. Dieses Feldelement ist ein Datensatz vom Typ 'Eintrag'. 'stadt' und 'leute' in den Zeilen 27 und 28 bzw. 29 nehmen auf die Datenfelder innerhalb dieses einen ausgewählten Datensatzes Bezug (vgl. Bild 2.1).
In Zeile 27 wird die Prozedur 'TextFrage' des Bausteins 'Fragen' aufgerufen. Im ersten Parameter wird als auszuschreibender Anforderungstext die Zeichenkette

```
"   Stadt"
```

an die Prozedur übergeben. Die Gänsefüsschen gehören nicht zur Zeichenkette; sie müssen davor und dahinter stehen. Der zweite Parameter legt fest, dass 'TextFrage' die eingelesene Zeichenkette im Datenfeld 'stadt' des Datensatzes 'tabelle[position]' ablegen soll. Nach Beendigung der Arbeit von 'TextFrage' geht es in Zeile 28 weiter.

```
stadt[0]
```

bezeichnet das unter der Nummer 0 gespeicherte Zeichen der Zeichenkette mit dem Namen 'stadt'. Entsprechend Zeile 14 ist dies das erste Zeichen. Falls die mittels 'TextFrage' eingelesene Zeichenkette mit einem Doppelkreuz beginnt, wird die Dialogeingabe abgebrochen (EXIT).
Der Aufruf von 'ZahlFrage' in Zeile 29 funktioniert analog zum Aufruf von 'TextFrage'. Die für die Parameter 'untereGrenze' und 'obereGrenze' an 'ZahlFrage' übermittelten Werte legen fest, dass nur Einwohnerzahlen zwischen 1 und 30000 akzeptiert werden.

11. Die sortierte Ausgabe der in 'tabelle' gesammelten Daten ist als "FOR-" oder "Laufanweisung" programmiert. Die Zeilen 36 und 46 legen fest, dass die Zeilen 37 bis 45 für verschiedene Werte der Variablen 'schluss' abzuarbeiten sind: Der Anfangswert von 'schluss' ist durch

    position-1

    gegeben; er soll dann bis zum Erreichen von 0 schrittweise um 1 verringert werden (Zeile 36). Für jeden Wert von 'schluss' geschehen drei Dinge. In den Zeilen 37 bis 42 wird unter den Tabelleneinträgen mit den Nummern 0 bis 'schluss' derjenige mit der grössten Einwohnerangabe 'leute' herausgesucht. In den Zeilen 43 und 44 wird dieser Eintrag ausgeschrieben. Die Wertzuweisung von Zeile 45 ersetzt den herausgesuchten und ausgegebenen, also fertig verarbeiteten Eintrag durch den derzeit letzten Eintrag. Zeile 45 schafft die Voraussetzungen für die Weiterarbeit ab Zeile 37: Wegen des um eins verringerten Werts von 'schluss' wird ein verkürztes Anfangsstück von 'tabelle' durchsucht!
12. In Zeile 37 wird 'tabelle[0]' zum "Kandidaten" ('maxNr') für den maximalen Eintrag ('maxLeute') gemacht.

    tabelle[0].leute

    bezeichnet das Datenfeld 'leute' im Datensatz 'tabelle[0]'. Diese Schreibweise tritt auch noch in Zeile 44 auf. In der durch die Zeilen 38 und 42 begrenzten Laufanweisung wird der Reihe nach geprüft, ob einer der Einträge mit den Nummern 1 bis 'schluss' eine grössere Einwohnerzahl enthält als gerade unter 'maxLeute' gemerkt ist. Jedesmal, wenn dieser Fall eintritt, wird der betreffende Eintrag zum neuen Maximumkandidaten gemacht (Wertzuweisungen in Zeile 40).
13. 'WriteInt', 'WriteString' und 'WriteLn' in den Zeilen 43 und 44 schreiben einen INTEGER-Wert, eine Zeichenkette bzw. ein Zeilenende. Der vor der äusseren Laufanweisung stehende Aufruf der Prozedur 'OpenOutput' (Zeile 35) bewirkt, dass alle diese Schreiboperationen auf das File "wohner.tab" führen. Nachdem die erfassten Daten vollständig ausgeschrieben sind, schliesst 'CloseOutput' (Zeile 47) die Ausgabe ab. Auf der nächsten Zeile steht das END des Programmoduls 'Staedte': Die Arbeit ist beendet.

Detaillierte Angaben zur Bedeutung der fünf vom Baustein 'InOut' importierten Bezeichner (Zeilen 9 und 10) und zu den erforderlichen Parameterangaben enthält der entsprechende Definitionsteil. Er ist im folgenden Abschnitt 2.4 abgedruckt.

2.4. Der Bibliotheksbaustein 'InOut'

Der Baustein 'InOut' stellt Eingabe- und Ausgabeoperationen über Terminal oder Textfile für Zeichenketten und ganze Zahlen zur Verfügung. Er geht auf einen Vorschlag des Schöpfers von MODULA-2, Niklaus Wirth, zurück [Wirt80] [Wirt82].
Ich werde die von 'InOut' exportierten Operationen in vielen kleinen Programmbeispielen benutzen. Die in dem als Programm 2.3 abgedruckten Definitionsmodul enthaltenen Kommentare sollten sowohl für das Verständnis der verwendeten Operationen als auch für die aktive Anwendung ausreichend sein.

```
 1 DEFINITION MODULE InOut;
 2 (*--------------------------------------------------------------
 3   Formatierte Ein-/Ausgabe.
 4 --------------------------------------------------------------*)
 5 EXPORT QUALIFIED
 6   Done, termCH,
 7   OpenOutput, CloseOutput, OpenInput, CloseInput,
 8   Read, ReadString, ReadInt,
 9   Write, WriteLn, WriteString, WriteInt,
10   WriteCard, WriteOct, WriteHex;
11 (*--------------------------------------------------------------*)
12 VAR Done: BOOLEAN;
13 (* 'Done' wird durch verschiedene Prozeduren gesetzt. Wurde die
14    Operation erfolgreich ausgefuehrt, so wird 'Done' auf TRUE
15    gesetzt, sonst auf FALSE.                                  *)
16 VAR termCH: CHAR;
17 (* Endezeichen mit dem die Eingabe bei 'ReadString' und
18    'ReadInt' abgeschlossen wurde.
19 --------------------------------------------------------------*)
```

```
20 PROCEDURE OpenOutput (fn: ARRAY OF CHAR);
21 (* Eroeffnen einer Datei fuer die Ausgabe.
22    Eing.par.:     fn      Datei-Spezifikation der zu
23                           eroeffnenden Datei.
24    War 'OpenOutput' erfolgreich, so ist 'Done' gleich TRUE, und
25    bis zum Aufruf von 'CloseOutput' werden die Ausgabedaten auf
26    diese Datei geschrieben.                                  *)
27 PROCEDURE CloseOutput;
28 (* Abschliessen der augenblicklichen Ausgabe-Datei.
29    Nachfolgende Ausgabedaten werden wieder ueber das Terminal
30    ausgeschrieben.                                           *)
31 PROCEDURE OpenInput (fn: ARRAY OF CHAR);
32 (* Eroeffnen einer Datei fuer die Eingabe.
33    Eing.par.:     fn      Datei-Spezifikation der zu
34                           eroeffnenden Datei.
35    War 'OpenInput' erfolgreich, so ist 'Done' gleich TRUE, und
36    bis zum Aufruf von 'CloseInput' werden die Eingabedaten von
37    dieser Datei gelesen.                                     *)
38 PROCEDURE CloseInput;
39 (* Abschliessen der augenblicklichen Eingabe-Datei.
40    Nachfolgende Eingabedaten werden wieder vom Terminal einge-
41    lesen.                                                    *)
42 PROCEDURE Read (VAR ch: CHAR);
43 (* Liest das naechste Zeichen vom aktuellen Eingabegeraet.
44    Ausg.par.:     ch     das gelesene Zeichen
45    'Done' ist TRUE, solange nicht das Ende der Datei erreicht
46    wurde.                                                    *)
47 PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
48 (* Liest eine Zeichenkette vom aktuellen Eingabegeraet.
49    Ausg.par.:     s       die eingelesene Zeichenkette,
50                           ohne das Endezeichen.
51    Fuehrende Leerzeichen werden akzeptiert und uebergangen.
52    Danach werden Zeichen in s eingelesen, bis ein erneutes
53    Leerzeichen oder irgendein Steuerzeichen (z.B. <esc>)
54    erkannt wird. 'ReadString' schneidet die Eingabezeichenkette
55    ab, wenn sie fuer s zu lang ist. 'termCH' wird mit dem
56    Endezeichen belegt. Werden die Eingabedaten vom Terminal
57    gelesen, so sind <bs> und <del> zum Editieren erlaubt.    *)
```

```
58 PROCEDURE ReadInt (VAR x: INTEGER);
59 (* Liest eine INTEGER-Zahl vom aktuellen Eingabegeraet.
60    Ausg.par.:    x     der gelesene Wert.
61    'ReadInt' arbeitet analog 'ReadString', aber die
62    Zeichenkette wird (falls moeglich) in einen INTEGER-Wert
63    uebergefuehrt. Dabei ist folgende Syntax zugelassen:
64         ["+"|"-"]digit{digit}.
65    'Done' ist TRUE, falls eine Konvertierung durchgefuehrt
66    werden konnte.                                          *)
67 PROCEDURE Write (ch: CHAR);
68 (* Schreibt ein Zeichen ueber das aktuelle Ausgabegeraet aus.
69    Eing.par.:    ch    das zu schreibende Zeichen          *)
70 PROCEDURE WriteLn;
71 (* Schreibt den Code fuer 'neue Zeile' ueber das aktuelle
72    Ausgabegeraet aus.                                      *)
73 PROCEDURE WriteString (s: ARRAY OF CHAR);
74 (* Gibt eine Zeichenkette ueber das aktuelle Ausgabegeraet aus.
75    Eing.par.:    s     die zu schreibende Zeichenkette.    *)
76 PROCEDURE WriteInt (x: INTEGER;
77                     n: CARDINAL);
78 (* Schreibt eine nach rechts ausgerichtete INTEGER-Zahl ueber
79    das aktuelle Ausgabegeraet in dezimaler Form aus.
80    Eing.par.:    x     der zu schreibende Wert.
81                  n     minimale Anzahl von Zeichen, die zur
82                        Darstellung benutzt werden sollen.
83    Die dezimale Darstellung der Zahl 'x' (einschliesslich '-',
84    falls 'x' negativ ist) wird ausgeschrieben. Werden fuer die
85    Darstellung von 'x' weniger als 'n' Zeichen benoetigt, so
86    werden links Leerzeichen eingefuegt. Falls es zur
87    Darstellung von 'x' erforderlich ist, werden mehr als 'n'
88    Zeichen ausgegeben.                                     *)
89 PROCEDURE WriteCard (x, n: CARDINAL);
90 (* Schreibt eine nach rechts ausgerichtete CARDINAL-Zahl ueber
91    das aktuelle Ausgabegeraet in dezimaler Form aus.
92    Eing.par.:    x     der zu schreibende Wert.
93                  n     minimale Anzahl von Zeichen, die zur
94                        Darstellung benutzt werden sollen.
95    Die dezimale Darstellung der Zahl 'x' wird ausgeschrieben.
```

```
 96    Werden fuer die Darstellung von 'x' weniger als 'n' Zeichen
 97    benoetigt, so werden links Leerzeichen eingefuegt. Falls es
 98    zur Darstellung von 'x' erforderlich ist, werden mehr als
 99    'n' Zeichen ausgegeben.                                    *)
100 PROCEDURE WriteOct (x, n: CARDINAL);
101 (* Schreibt eine CARDINAL-Zahl in oktaler Form ueber das
102    aktuelle Ausgabegeraet aus.
103    Eing.par.:    x     der zu schreibende Wert.
104                  n     minimale Anzahl von Zeichen, die zur
105                        Darstellung benutzt werden sollen.    *)
106 PROCEDURE WriteHex (x, n: CARDINAL);
107 (* Schreibt eine CARDINAL-Zahl in hexadezimaler Form ueber das
108    aktuelle Ausgabegeraet aus.
109    Eing.par.:    x     der zu schreibende Wert.
110                  n     minimale Anzahl von Zeichen, die zur
111                        Darstellung benutzt werden sollen.
112 ------------------------------------------------------------*)
113 END InOut.
```

Programm 2.3. Definitionsteil des Bausteins 'InOut' (gekürzt)

Programm 2.3 enthält alle im Hauptprogramm 'Staedte' und im Baustein 'Fragen' verwendeten Bezeichner der Robotron-Version von 'InOut' [Robo87b]. Diese Version stimmt weitgehend mit der Logitech-Fassung [Logi85] von 'InOut' überein.

Der Baustein ist so eingerichtet, dass anfangs alle 'Read'-Prozeduren vom Terminal lesen und alle 'Write'-Prozeduren auf das Terminal schreiben. Eine "Umlenkung" der Ausgabe auf ein File wird durch 'OpenOutput' erreicht; 'CloseOutput' stellt die Ausgabe wieder auf Terminal um (s. Zeilen 20 bis 30). Für die Eingabe existieren analoge Möglichkeiten (s. Zeilen 31 bis 41).

Auf Zeile 64 ist die Syntax der von 'ReadInt' akzeptierten vorzeichenbehafteten ganzen Zahlen in EBNF notiert. Diese Darstellungsform wird im Abschnitt 4.1 ausfürlich erläutert. Der bisher noch nicht verwendete Standardtyp CARDINAL ist im Abschnitt 6 beschrieben. Hinweise auf die Oktal- und Hexadezimaldarstellung von Zahlen enthält Abschnitt 4.2.

2.5. Implementation des Bausteins 'Fragen'

Programm 2.1 zeigte den Definitionsteil des Bausteins 'Fragen'. Er enthält zwei Prozedurköpfe. Der Baustein ist erst dann vollständig programmiert, wenn auch die zu diesen Köpfen gehörenden Prozeduren vorliegen: Die im Definitionsteil eines Bausteins versprochenen Dinge muessen in einem Implementationsteil realisiert werden. Die Verbindung zwischen beiden Teilen wird durch die Angabe desselben Bausteinnamens erreicht. Der Implementationsteil eines Bausteins heisst in der Terminologie der Sprache MODULA-2 Implementationsmodul. Es folgen Erläuterungen zu Programm 2.4, das einen Implementationsmodul zum Baustein 'Fragen' zeigt.

1. Ein Implementationsmodul beginnt mit einem Kopf (Zeile 1), der aus den Schlüsselwörtern IMPLEMENTATION und MODULE, dem Bausteinnamen sowie einem Semikolon besteht. Auch hier stehen am Modulende ein END, der Bausteinname und ein Punkt (Zeile 32).
2. 'Fragen' importiert fünf Bezeichner vom Standardbaustein 'InOut' (Zeilen 6 und 7), deren Bedeutung Programm 2.3 zu entnehmen ist.
3. Die Prozedurköpfe in Definitions- und Implementationsmodul eines Bausteins müssen übereinstimmen. Vor dem Anweisungsteil der Prozedur 'TextFrage' steht das Schlüsselwort BEGIN (Zeile 11). Auf den folgenden zwei Zeilen werden die beiden Parameter 'anforderung' und 'text' an die 'InOut'-Prozeduren 'WriteString' bzw. 'ReadString' weitergereicht. Die Prozedur wird mit END, dem Prozedurnamen und einem Semikolon abgeschlossen (Zeile 15).
4. Vor dem Anweisungsteil zu 'ZahlFrage' ist in den Zeilen 20 und 21 eine lokale Variable 'i' vereinbart. Man nennt sie "lokal", weil sie nur innerhalb der Prozedur 'ZahlFrage' - d. h. bis zum END in Zeile 30 - benutzbar ist.
5. Nach dem Ausschreiben der 'anforderung' mit nachgestelltem Doppelpunkt (Zeile 24) liest 'ReadInt' eine ganze Zahl ein. Der gelesene Wert erscheint in der lokalen Variablen 'i' (Zeile 25). Die Wertzuweisung an den Referenzparameter 'zahl' (Zeile 29) liefert das Resultat der Arbeit von 'ZahlFrage' nach "aussen".

```
 1 IMPLEMENTATION MODULE Fragen;
 2 (*-------------------------------------------------------------
 3    Prozeduren zur Dialogeingabe von Zeichenketten und ganzen
 4    Zahlen
 5 -------------------------------------------------------------*)
 6 FROM InOut IMPORT
 7   WriteString, WriteLn, ReadString, ReadInt, Done;
 8 (*-------------------------------------------------------------*)
 9 PROCEDURE TextFrage(anforderung: ARRAY OF CHAR;
10                     VAR text: ARRAY OF CHAR);
11 BEGIN
12   WriteString(anforderung); WriteString("' : ");
13   ReadString(text);
14   WriteLn;
15 END TextFrage;
16
17 PROCEDURE ZahlFrage(anforderung: ARRAY OF CHAR;
18                     untereGrenze,obereGrenze: INTEGER;
19                     VAR zahl: INTEGER);
20 VAR
21   i: INTEGER;
22 BEGIN
23   REPEAT
24     WriteString(anforderung); WriteString(" : ");
25     ReadInt(i);
26     WriteLn;
27   UNTIL Done AND
28         (i >= untereGrenze) AND (i <= obereGrenze);
29   zahl := i;
30 END ZahlFrage;
31 (*-------------------------------------------------------------*)
32 END Fragen.
```

Programm 2.4. Implementationsteil des Bausteins 'Fragen'

6. Das in den Zeilen 11 bis 14 des Definitionsmoduls 'Fragen' (Programm 2.1) geforderte Verhalten bei fehlerhaften bzw. ausserhalb von 'untereGrenze' und 'obereGrenze' liegenden Eingaben wird durch die Einbettung der Zeilen 24 bis 26 in eine REPEAT-UNTIL-Schleife erreicht. Die zwischen den Schlüsselwörtern REPEAT ("wiederholen", engl.) und UNTIL ("bis", engl.) stehende Anweisungsfolge muss so lange wiederholt werden, bis die hinter UNTIL notierte Bedingung

   Done AND (i>=untereGrenze) AND (i<=obereGrenze)

   erfüllt ist. Das Schlüsselwort AND entspricht dem logischen "und" zur Verknüpfung von Wahrheitswerten, d. h., die dreiteilige Bedingung ist genau dann erfüllt, wenn alle ihre Teile erfüllt sind.
   Die von 'InOut' importierte Variable 'Done' ("gemacht", engl.) hat den Standardtyp 'BOOLEAN'. Der durch 'BOOLEAN' beschriebene Wertebereich besteht aus 'TRUE' ("wahr", engl.) und 'FALSE' ("falsch", engl.). 'Done' zeigt an, ob 'ReadInt' eine Zahl eingelesen hat oder nicht (vgl. Programm 2.3, Zeilen 9 bis 12 sowie 46 und 47).
   Das Ausschreiben der 'anforderung' mit nachfolgendem Lesen einer ganzen Zahl wird also so lange wiederholt, bis wirklich eine Zahl gelesen wurde und diese Zahl in dem geforderten Bereich liegt. Im Normalfall ("richtige" Antwort) ist die Bedingung gleich beim erstenmal erfüllt, und der Nutzer am Terminal merkt nichts von dem Zyklus.
7. Für das einwandfreie Funktionieren dieser Implementation des Bausteins 'Fragen' muss gefordert werden, dass weder die Eingabe noch die Ausgabe des Bausteins 'InOut' durch Rufe von 'OpenInput' bzw. 'OpenOutput' umgelenkt ist. Bei der Benutzung von 'Fragen' in 'Staedte' ist diese Bedingung erfüllt. Die Ausgabeumlenkung erfolgt erst nach Abschluss der Dialogarbeit.

Damit ist die Präsentation eines kleinen MODULA-2-Programms abgeschlossen. Jetzt müsste man das Programm ausprobieren können. Genau das ist das Ziel des nächsten Abschnitts.

3. Vom Quelltext zur Programmabarbeitung

Der Weg vom Quelltext auf Papier bis zu einem abarbeitungsfähigen Programm im Rechner führt über mehrere "Hilfsprogramme" (Editor, Compiler, Linker). Und wenn es dann soweit ist, dann sollte man mit dem wohldurchdachten Testen des Programms beginnen.
Ich fange mit einem der kürzesten "sinnvollen" MODULA-2-Programme an. Es benutzt keine privaten Bausteine. Anschliessend dokumentiere und diskutiere ich die Inbetriebnahme des im Abschnitt 2 entwickelten Programms 'Staedte'. Abschnitt 3.3 befasst sich mit dem systematischen Testen dieses Programms.

3.1. Die Schritte bis zur Abarbeitung eines MODULA-2-Programms

Bevor ein in Ihrem Kopf (oder besser auf einigen Blättern Papier) vorliegendes MODULA-2-Programm abgearbeitet werden kann, sind folgende drei Arbeitsgänge erforderlich:

1. Das Programm muss in den Rechner eingegeben werden. Hierzu eignet sich ein üblicher Texteditor. Im Resultat entstehen Files, die den Programmtext enthalten. Wir wollen diese Files im folgenden als "Quelltextfiles" bezeichnen. Für jeden Definitionsteil, jeden Implementationsteil und jeden Programmodul ist je ein Quelltextfile anzulegen.
2. Jedes einzelne Quelltextfile muss vom MODULA-2-Compiler bearbeitet - d. h. übersetzt - werden.
3. Der übersetzte Programmodul muss mit den übersetzten Implementationsteilen aller jener Bausteine, von denen er importiert hat, verbunden werden. Wenn die Bausteine von weiteren Bausteinen importieren, dann sind diese einzubeziehen usw. Im Resultat entsteht ein File, welches das MODULA-2-Programm in einer abarbeitbaren Form enthält.
   Ausgeführt werden diese Arbeiten von einem sogenannten Programmverbinder oder "Linker".

Der unter Punkt 1 benötigte Editor und seine Bedienung sind eine Angelegenheit des jeweiligen Betriebssystems. Ich verzichte deshalb hier auf weitere Angaben.

Alle mir bekannten MODULA-2-Systeme weisen bez. der Bedienung von Compiler und Linker und bez. der Abarbeitung fertiger Programme Unterschiede auf. Das auf dem Bildschirm zu beobachtende Erscheinungsbild beim Compilieren und Linken folgt aber einem einheitlichen Muster. Die im weiteren angegebenen Protokolle zeigen das Wesen der Abläufe und sollten die Orientierung in beliebigen Programmierumgebungen für MODULA-2 ermöglichen. Sie entsprechen weitgehend dem MODULA-2-System M2SCP der Humboldt-Universität zu Berlin für 8-Bit-Rechner mit CP/M-kompatiblen Betriebssystemen [Blac85]. M2SCP basiert auf der "klassischen" Züricher Implementation M2RT11 für PDP-11-Rechner [Geis81]. M2RT11 ist weltweit als Vorbild der bewährten MODULA-2-Systeme für moderne 16- und 32-Bit-Rechner anzusehen.
Gegeben sei das Programm 3.1. Ich setze voraus, dass es unter dem Filenamen 'hier.mod' auf einer Diskette gespeichert ist und dass diese Diskette keine weiteren Files enthält. Was dieses Programm macht, das sollte jeder Leser sofort verstehen bzw. mit Hilfe der Erläuterungen zum Baustein 'InOut' (s. Programm 2.3) problemlos herausfinden können.

```
1 MODULE Hier;
2 FROM InOut IMPORT
3   WriteString, WriteLn;
4 BEGIN
5   WriteString("Hier bin ich!");
6   WriteLn;
7 END Hier.
```

Programm 3.1. Programmodul 'Hier'

Auf der Ebene des Betriebssystems sei 'A' als "aktuelles Laufwerk" eingestellt. Die Diskette mit dem File 'hier.mod' liege in diesem Laufwerk. Ich übersetze den Programmodul 'Hier', verbinde ihn mit den notwendigen Bausteinen und arbeite ihn ab. Durch Aufrufe des Kommandos 'dir' stelle ich fest, welche Files auf dem Laufwerk A entstehen. Auf dem Bildschirm spielt sich folgendes ab (das, was ich eingetippt habe, ist unterstrichen; alles andere produzieren das Betriebssystem und die Programme des MODULA-2-Systems).

```
A>dir
  hier.mod
A>m2c
 source file> hier.mod
p1 syntax analysis
 InOut: SY:InOut.SYM
p2 declaration analysis
p3 block analysis
p4 code generation
end compilation
A>dir
  hier.mod
  hier.obj
  hier.ref
A>m2l
 master file> hier
  link files:
 InOut: SY:InOut.OBJ
 TTIO: SY:TTIO.OBJ
 Storage: SY:Storag.OBJ
end linkage
A>dir
  hier.mod
  hier.obj
  hier.ref
  hier.lod
A>m2 hier
Hier bin ich!
A>
```

Der MODULA-2-Compiler (Kommando 'm2c') meldet sich mit der Frage nach dem zu übersetzenden Quelltextfile. Als Reaktion auf die Eingabe des Filenamens erscheinen laufend Hinweise über den Fortgang der Arbeiten ('p1 ...' bis 'end compilation'). Der Compiler wertet zuerst die Importlisten aus und sucht auf dem aktuellen Laufwerk aufbereitete Angaben über importierte Bausteine. Wenn da nichts zu finden ist, dann wird in der Bausteinbibliothek nachgesehen. Diese Bausteinbibliothek ist in meinem Beispiel unter dem symbolischen Gerätenamen 'SY' erreichbar. Das Protokoll enthält

die Mitteilung, dass die Information zum Baustein 'InOut' dem File 'SY:InOut.SYM' aus der Bausteinbibliothek entnommen wurde. Als Resultat der Compilation sind zwei neue Files entstanden: 'hier.obj' und 'hier.ref'.
Der MODULA-2-Linker (Kommando 'm2l') fragt sofort nach dem Namen eines Files, das durch die Compilation eines Programmoduls entstanden ist. Ich antworte mit 'hier' und verlasse mich darauf, dass der Linker "weiss", wie diese keinen Punkt enthaltende Zeichenkette zu dem vom Compiler gelieferten Filenamen zu ergänzen ist. Der Linker trägt alle jene Bausteine zusammen, von denen das Hauptprogramm direkt oder indirekt importierte, und informiert darüber.
Wir sehen, dass erwartungsgemäss der Baustein 'InOut' aus der Bibliothek geholt wurde (File 'SY:InOut.OBJ'). Unerwartet treten weitere Bausteine auf: 'Storage' und 'TTIO'. Beide Bausteine werden offenbar im Implementationsteil von 'InOut', der dem Nutzer i. allg. unbekannt sein wird, importiert. Das ist nicht beunruhigend, solange die entsprechenden Files in der Bibliothek vorhanden sind. Beim Linken des Programms 'Hier' auf kommerziellen MODULA-2-Systemen für 16- oder 32-Bit-Rechner werden etwa ein Dutzend Bausteine aus der Bibliothek geholt. Was diese Bausteine machen, ist für den Nutzer völlig uninteressant.
Die gemischte Benutzung von Gross- und Kleinbuchstaben in Filenamen ist bei den mir bekannten MODULA-2-Systemen üblich. In der Regel werden diese Namen von der Fileverwaltung des Betriebssystems "eingeebnet": Intern wird alles in Gross- oder alles in Kleinbuchstaben verwandelt.
Das vom Linker erzeugte File 'hier.lod' kann nun mit der Kommandozeile 'm2 hier' abgearbeitet werden. Analog zum Linker "weiss" auch das Programm 'm2', dass die keinen Punkt enthaltende Zeichenkette 'hier' noch zu dem vom Linker erzeugten Filenamen ergänzt werden muss. Es zeigt sich das gewünschte Ergebnis: Die Ausschrift

```
'Hier bin ich!
```

erscheint auf dem Bildschirm. Anschliessend ist das Betriebssystem bereit zur Entgegennahme weiterer Kommandos.

Sie, lieber Leser, sollten jetzt auf Ihrem Rechner diese Schritte nachvollziehen! Wahrscheinlich funktioniert bei Ihrem MODULA-2-

System alles ein wenig anders. Die folgenden Bemerkungen sollen Ihnen die Orientierung erleichtern.

1. Die Kommandonamen ('dir', 'm2c', 'm2l', 'm2') sind nicht einheitlich. Auch die Konventionen zur Bildung von Filenamen unterscheiden sich. Die "richtigen" Namen finden Sie in den Handbüchern zu Ihrer Software.
2. Die vom Compiler und vom Linker ausgegebenen Arbeitsprotokolle sind i. allg. umfangreicher. Fast keine Firma verzichtet auf ein paar Zeilen mit ihrem Namen und ihrer Adresse sowie einer genauen Kennzeichnung der Version des MODULA-2-Systems. Die Angaben über Zugriffe auf Bausteinbibliotheken sind oft ausführlicher.
   Die mit 'p1', ..., 'p4' beginnenden Zeilen können auch fehlen, z. B. bei sogenannten Einpasscompilern.
3. Bei vielen Systemen kann/muss beim Aufruf des Compilers bzw. des Linkers der Name des zu behandelnden Files gleich mit in der Kommandozeile angegeben werden.
4. Nicht alle Linker sind in der Lage, selbständig alle benötigten Bausteine zu finden. Bisweilen müssen neben dem Filenamen des übersetzten Hauptprogramms auch noch die Filenamen aller Bausteine und/oder Filenamen von Bausteinbibliotheken angegeben werden. Linker, die speziell an die Sprache MODULA-2 gebunden sind und mit einem MODULA-2-System mitgeliefert werden, lösen normalerweise Importe selbständig auf. Der andere Fall tritt dann ein, wenn auch für MODULA-2-Programme ein zum Betriebssystem gehörender "allgemeiner" Linker Verwendung findet.
5. Bei der Arbeit des Compilers werden i. allg. mehrere temporäre Files für Zwischenergebnisse angelegt. "Temporär" bedeutet, dass diese Files nach Beendigung der Compilation nicht mehr benötigt und normalerweise automatisch gestrichen werden. Zwei für Neulinge verwunderliche Dinge können auftreten. Wenn der Speicherplatz für diese Hilfsfiles nicht ausreicht, dann bricht der Compiler seine Arbeit ab. Man muss an der richtigen Stelle (s. das jeweilige Systemhandbuch) einige entbehrliche Files löschen, und schon funktioniert alles. Nach einem irregulären Ende einer Compilation stösst man auf Files, deren Herkunft unklar ist. Oft handelt es sich hier um solche tempo-

räre Files des Compilers, die nicht automatisch gestrichen wurden.

6. In meinem Beispiel auf S. 37 wurden sowohl der Compiler als auch der Linker auf der Betriebssystemebene gestartet. Es gibt Fälle, wo man ein MODULA-2-System aufruft und alles weitere dann innerhalb dieses Systems abläuft. Die Kommandos zum Aufruf des Compilers/Linkers sind hier nicht an das Betriebssystem gerichtet, sondern an das MODULA-2-System. Manchmal kann man zwischen beiden Varianten wählen.
7. Die Abarbeitung des vom Linker erstellten Files 'hier.lod' wurde mittels der Kommandozeile

   m2 hier

   veranlasst. Das File 'hier.lod' enthält kein sofort auf der Betriebssystemebene ausführbares Programm; man benötigt eine kleine "Starthilfe" - das Programm 'm2'. Bei MODULA-2-Systemen, die einen allgemeinen, zum Betriebssystem gehörenden Linker verwenden (vgl. Bem. 4), entstehen sofort Programme, die auf der Betriebssystemebene abarbeitbar sind. Alle anderen MODULA-2-Systeme bieten ein zusätzliches Hilfsprogramm an, das aus einem File wie 'hier.lod' ein direkt abarbeitungsfähiges Programmfile macht. Dieses Werkzeug wird i. allg. erst bei der Überführung von MODULA-2-Programmen in den Routinebetrieb wirklich benötigt.

Manche MODULA-2-Systeme bieten einen sogenannten syntaxorientierten Editor an, der speziell auf die Sprache MODULA-2 zugeschnitten ist, z. B. [Logi85]. Der Aufwand, den Umgang mit einem solchen Programmierwerkzeug zu erlernen, lohnt sich.

3.2. Inbetriebnahme des Programms 'Staedte'

Wenn es Ihnen gelungen ist, den Programmodul 'Hier' auf Ihrem Rechner abzuarbeiten, dann sollten Sie auch die Quelltexte des Bausteins 'Fragen' (Programme 2.1 und 2.4) sowie den Programmodul 'Staedte' (Programm 2.2) aus Abschnitt 2 eintippen, übersetzen, linken und abarbeiten. Das folgende Terminalprotokoll zeigt den prinzipiellen Ablauf. Es ist aus Platzgründen zweispaltig wiedergegeben.

```
A>dir
  fragen.def
  staedt.mod
  fragen.mod
A>m2c
 source file> fragen.def
p1 syntax analysis
p2 declaration analysis
symbol file generation
end compilation
A>dir
  fragen.def
  staedt.mod
  fragen.mod
  fragen.sym
A>m2c
 source file> staedt
p1 syntax analysis
 Fragen: A:Fragen.SYM
 InOut: SY:InOut.SYM
p2 declaration analysis
p3 block analysis
p4 code generation
end compilation
A>m2c
 source file> fragen
p1 syntax analysis
 Fragen: A:Fragen.SYM
 InOut: SY:InOut.SYM
p2 declaration analysis
p3 block analysis
p4 code generation
end compilation
```

```
A>m2l
master file> staedt
 link files:
  Fragen: A:Fragen.OBJ
  InOut: SY:InOut.OBJ
  TTIO: SY:TTIO.OBJ
  Storage: SY:Storag.OBJ
 end linkage
 A>dir
   fragen.def
   staedt.mod
   fragen.mod
   fragen.sym
   staedt.obj
   staedt.ref
   fragen.obj
   fragen.ref
   staedt.lod
 A>m2 staedt
    Stadt : Elsterwerda
    Einwohnerzahl : 10500
    Stadt : Wahrenbrueck
    Einwohnerzahl : 650
    Stadt : Doberlug-Kirchhain
    Einwohnerzahl : 0
    Einwohnerzahl : 9200
    Stadt : #
 A>dir *.tab
   wohner.tab
 A>
```

Drei Bemerkungen zu den in diesem Terminalprotokoll sichtbaren Besonderheiten:

1. Bei der Bearbeitung eines Definitionsteils verhält sich der Compiler anders als sonst. Im Resultat der Übersetzung von

'fragen.def' entsteht ein File 'fragen.sym'. Dieses File wird vom Compiler sowohl bei der Behandlung der Importe des Programmoduls 'Staedte' als auch bei der Bearbeitung des Implementationsteils 'Fragen' herangezogen. Man sieht es an den Protokollzeilen 'Fragen: A:Fragen.SYM'.

2. Beim zweiten und dritten Aufruf des Compilers sind die nach der entsprechenden Anfrage eingetippten Namen der Quelltextfiles verkürzt. Wenn dem Compiler ein Name angeboten wird, der keinen Punkt enthält, dann ergänzt er selbständig die vier Zeichen '.mod'.
3. Aus den ursprünglich vorhandenen drei Files sind bis zur Abarbeitung neun Files hervorgegangen! Und dann entsteht noch das File 'wohner.tab'.

Der Datenfluss beim Compilieren und Linken von MODULA-2-Programmen, die sich aus Bausteinen zusammensetzen, und die in diesem Zusammenhang von Compiler und Linker verfolgte Strategie zur Bildung von Filenamen sind Gegenstand der Abschnitte 10.2 und 10.3.

Das Terminalprotokoll von der vorigen Seite ist unrealistisch. Mit grosser Wahrscheinlichkeit werden sich beim Eingeben der Quelltexte Tippfehler einschleichen. Als Folge davon wird der Compiler Fehler finden und darauf reagieren. Sie müssen die Fehler mittels eines Texteditors korrigieren und dann erneut den Compiler rufen usw., bis keine Fehlernachrichten mehr auftreten.
Die Unterrichtung des Nutzers über vom Compiler entdeckte Quelltextfehler ist auf zwei Wegen üblich:

1. Erzeugung eines Übersetzungsprotokolls, das den vollständigen Quelltext enthält, und/oder
2. Ausgabe eines Kurzprotokolls, das lediglich die bemängelten Zeilen und die zugehörigen Fehlernachrichten enthält.

Manche Compiler liefern als Fehlernachricht nur eine Nummer, unter der man dann in der Dokumentation nachschlagen muss; andere schreiben gleich einen Kurztext. Alle Compiler bemühen sich um eine möglichst genaue Anzeige des Fehlerorts.
Zu Demonstrationszwecken habe ich in den schon gut bekannten Programmodul 'Hier' (Programm 3.1) einen kleinen Fehler eingebaut (Quelltextfile 'hier.mod'). Beim Übersetzen ergibt sich folgender Ablauf.

```
A>dir
  hier.mod
A>m2c
 source file> hier.mod
p1 syntax analysis
 InOut: SY:InOut.SYM
p2 declaration analysis
 ---- error
p3 block analysis
 ---- error
listing generation
end compilation
A>dir
  hier.mod
  hier.lst
A>type hier.lst
     1  MODULE Hier;
     2  FROM InOut IMPORT
     3    WriteString, Writeln;
 *****                        ^ 71
     4  BEGIN
     5    WriteString("Hier bin ich!");
     6    WriteLn;
 *****          ^ 73
     7  END Hier.
A>
```

Der Compiler entdeckt in zwei Phasen seiner Arbeit Fehler. Es müssen deshalb mindestens zwei Fehler vorliegen; es können aber auch mehr sein. An Stelle der Mitteilungszeile

```
p4 code generation
```

erscheint jetzt ein Hinweis auf die Erzeugung eines Protokollfiles:

```
listing generation.
```

Wie dem danach erfragten Fileverzeichnis zu entnehmen ist, entstand ein File 'hier.lst'. Ich sehe mir dieses File auf dem Bildschirm an (Kommando 'type') und stelle fest: In Zeile 3 wurde bei der Verarbeitung des Bezeichners 'Writeln' Fehler 71 gefunden. In Zeile 6 ist beim Bezeichner 'WriteLn' Fehler 73 ausgewiesen. Nachschlagen in der Dokumentation des MODULA-2-Systems führt auf die folgenden

Mitteilungen:

71: identifier not exported from qualifying module

73: identifier not declared.

Fehler 71 besagt, dass der fragliche Bezeichner (das ist der, bei dem der Fehler angezeigt wird) von dem entsprechenden Modul (in unserem Fall 'InOut') nicht exportiert wird. 'Writeln' muss also ein Irrtum oder ein Schreibfehler sein. Ein Blick auf die Exportliste im Definitionsteil des Bausteins 'InOut' (Programm 2.3) zeigt, dass ein Tippfehler vorliegt: An Stelle des Grossbuchstabens 'L' habe ich den Kleinbuchstaben 'l' geschrieben.

Fehler 73 besagt: Der Bezeichner 'WriteLn' ist an dieser Programmstelle unbekannt; er wurde nicht ordnungsgemäss "vereinbart". Hier liegt ein tückischer, aber sehr häufiger sogenannter Folgefehler vor: Die durchaus richtige Verwendung eines von dem fraglichen Baustein auch wirklich exportierten Bezeichners wird infolge eines Fehlers in der Importliste als fehlerhaft zurückgewiesen. Wer das sofort durchschaut, der wird den Fehler in Zeile 3 mit Hilfe eines Texteditors reparieren und sicher sein, dass dadurch auch der Fehler in Zeile 6 verschwindet. Wer in solchen Situationen Zweifel hat oder die Ursache von angezeigten Fehlern nicht entdecken kann, dem sei folgende Verfahrensweise empfohlen:

> Korrigieren Sie alle jene Fehler, die Sie verstehen und zu denen Ihnen eine Reparatur einfällt! Dann übersetzen Sie erneut!

Das nun erscheinende Fehlerbild ist i. allg. klarer. Die Folgefehler der von Ihnen korrigierten Fehler sind nicht mehr vorhanden. Vielleicht ist der Quelltext auch plötzlich schon fehlerfrei.

Abschliessend sei noch darauf verwiesen, dass die MODULA-2-Compiler in der Regel eine Möglichkeit anbieten, um auch bei fehlerfreien Compilerläufen ein Übersetzungsprotokoll ('.lst'-File) zu erhalten. Häufig handelt es sich um sogenannte Schalter oder Optionen. Schalter/Optionen werden in der Kommandozeile zum Aufruf des Compilers oder bei der Antwort auf die Frage nach dem Namen des Quelltextfiles mit angegeben. Sie bestehen i. allg. aus einem führenden Divisionsstrich '/' und bestimmten Zeichen oder Zeichenketten. Die genaue Information muss unter der Rubrik "Schalter" oder "Optionen" in der Dokumentation des jeweiligen MODULA-2-Systems gesucht werden. Bei dem für die Beispiele in diesem Buch verwendeten Compiler

wird ein Übersetzungsprotokoll durch Anhängen des Schalters '/l' an den Quelltextfilenamen erzwungen, z. B.

source file> hier.mod/l

In der Regel gibt es weitere Schalter zur Steuerung der Arbeitsweise von Compiler und Linker, z. B. Linkprotokoll erzeugen.

3.3. Einige Testläufe des Programms 'Staedte'

Im vorigen Abschnitt wurde erläutert, wie man von den Quelltexten eines MODULA-2-Programms bis zum abarbeitungsfähigen Programm kommt. Um inhaltliche Fehler in dem Programm 'Staedte' zu finden, arbeite ich es mit ausgewählten Eingabedaten ab und vergleiche die Resultate mit den vorher aufgeschriebenen erwarteten Resultaten. Man spricht vom "Testen" des Programms und von "Testdaten". Zwei bekannte Allgemeinplätze zum Thema Programmtest lauten:

1. Durch Tests kann nur die Anwesenheit von Fehlern gezeigt, nicht aber deren Abwesenheit bewiesen werden.
2. Tests soll man systematisch und zielgerichtet planen und durchführen.

Die erste Feststellung ist tiefsinnig und sehr wahr. Durch umfangreiches Testen - einschliesslich "Reparatur" gefundener Fehler und erneutem Test - erhöht sich das Vertrauen in die Funktionsfähigkeit eines Programms; aber mehr nicht!

Zur Frage des systematischen Programmtests beschränke ich mich auf Bemerkungen über die Auswahl solcher Testdaten, die der Einschätzung des Verhältnisses von Eingabedaten und Resultaten dienen. Vier Gruppen von Tests sollten stets ausgeführt werden:

1. **Normalfälle:** Verhalten bei korrekten Eingabedaten, die der normalen geplanten Verwendung des Programms entsprechen.
2. **Randfälle:** Reaktion auf Randfälle bez. der erlaubten Eingabedaten bzw. Ermitteln, wo diese "Ränder" liegen.
3. **Unfug:** Programmverhalten bei "unvernünftigen" zufälligen Eingabedaten.
4. **Abbruch:** Resultate, wenn das Programm mitten in der Arbeit abbricht oder abgebrochen wird.

Ein Normalfalltest für das Programm 'Staedte' kann etwa so aussehen, wie am Ende des Terminalprotokolls auf Seite 41 gezeigt. Das

erwartete Resultat (File 'wohner.tab') ist klar (vgl. S. 17). Da man natürlich mehrere Testläufe dieser Kategorie durchführen soll, versuche ich es noch einmal und beginne mit der Stadt Bad Liebenwerda, die 6800 Einwohner hat.

```
A>m2 staedt
   Stadt : Bad
   Einwohnerzahl : Liebenwerda
   Einwohnerzahl : 6800
   Stadt : #
A>type wohner.tab
    6800  Bad
A>
```

Grosse Verwunderung! Ich hatte die Daten schnell eingetippt, ohne jedesmal auf die Reaktion des Programms zu achten, und sah hinterher, was sich getan hatte. Weitere Versuche bestätigen: Immer dann, wenn ich das zwischen 'Bad' und 'Liebenwerda' stehende Leerzeichen eintippe, erscheint die Eingabeaufforderung

```
Einwohnerzahl :
```

auf dem Bildschirm. Finden Sie heraus, warum das so ist! Betrachten Sie zuerst den Programmodul 'Staedte' (Programm 2.2), dann die interessierende Stelle im Implementationsteil des Bausteins 'Fragen' (Programm 2.4) und abschliessend den Kommentar zu dem betreffenden Prozedurkopf im Definitionsteil des Bausteins 'InOut' (Programm 2.3). Wenn Ihnen kein Denkfehler unterlaufen ist, dann haben sie die Erklärung in den Zeilen 52 bis 54 des Definitionsmoduls 'InOut' gefunden. Eine Programmkorrektur ist auf dem derzeitigen Stand Ihrer Einführung in die Sprache MODULA-2 nicht sinnvoll. Man könnte sich vorerst darauf einigen, das kritische Leerzeichen durch einen Unterstrich zu ersetzen: 'Bad_Liebenwerda'.

Auf diese Weise wurde "versehentlich" gleich ein Randfall bemerkt. Weitere Randfälle sind: sehr lange Namen, leere Namen, negative Einwohnerzahlen, Einwohnerzahlen groesser als 30000. Aber auch: zuviel Einträge (Was ist "zuviel"? 70 oder 71?), gar kein Eintrag (d. h. sofort '#' als Stadt angeben), mehrere Einträge mit demselben Namen und/oder derselben Einwohnerzahl.

Überlegen Sie sich, wie das Programm auf diese Dinge reagieren sollte, und probieren Sie es danach aus! Stellen Sie einen Katalog von Wünschen für ein sinnvolleres Programmverhalten zusammen.

Gewiss könnte man mehrere Einträge mit demselben Namen schon der Testkategorie Unfug zurechnen. Darüber hinaus meine ich mit Unfug Städtenamen wie 'Muehlberg000&3' oder '12345%=anna' und Einwohnerzahlen wie '10003,17', 'viele' oder 'xyzblahblah'. Wie verhält sich das Programm zu solchen Eingaben?
Die Frage nach den übrigbleibenden Resultaten bei Programmabbruch betrifft das File 'wohner.tab'. Das folgende Terminalprotokoll beginnt mit einem den Normalfall betreffenden Problem: Was geschieht mit dem File 'wohner.tab', wenn das Programm 'Staedte' mehrmals nacheinander abgearbeitet wird?

```
A>m2 staedt
   Stadt : Wahrenbrueck
   Einwohnerzahl : 650
   Stadt : Elsterwerda
   Einwohnerzahl : 10500
   Stadt : #
A>type wohner.tab
   10500  Elsterwerda
     650  Wahrenbrueck
A>m2 staedt
   Stadt : Bad Liebenwerda
   Einwohnerzahl : 6800
   Stadt : #
    6800  Bad_Liebenwerda
A>type wohner.tab
   10500  Elsterwerda
     650  Wahrenbrueck
```

```
A>ren wohner.tab w1.tab
A>m2 staedt
   Stadt : Bad Liebenwerda
   Einwohnerzahl : 6800
   Stadt : #
A>type wohner.tab
    6800  Bad_Liebenwerda
A>ren wohner.tab w2.tab
A>m2 staedt
   Stadt : Muehlberg
   Einwohnerzahl : 3600
   Stadt : #
^C
A>dir *.tab
  w1.tab
  w2.tab
A>
```

Nachdem die Eingabe bei der zweiten Abarbeitung des Programmoduls 'Staedte' durch ein Doppelkreuz beendet wurde, erscheint auf dem Bildschirm eine Zeile, die eigentlich in das File 'wohner.tab' geschrieben werden sollte. Schuld daran ist die Prozedur 'OpenOutput' aus dem Baustein 'InOut' in der Bausteinbibliothek: Wenn man von ihr verlangt, ein bereits existierendes File zum Schreiben zu eröffnen, dann lehnt sie das ab und tut gar nichts, d. h., die Ausgabe wird nicht auf das gewünschte File "umgelenkt" und erscheint weiter auf dem Terminal. Diese Verhaltensweise ist den

Kommentaren zum Definitionsteil 'InOut' (Programm 2.3) nicht zu entnehmen. Man kann sie nur durch ein Experiment herausbekommen, oder man verfügt über weitere Informationen zum Baustein 'InOut', z. B. den Implementationsteil. In anderen MODULA-2-Systemen sind durchaus andere Reaktionen möglich. Ich kenne folgende Varianten:

- Löschen des alten Inhalts von 'wohner.tab'.
- Anfügen der neuen Ausgaben an das alte File 'wohner.tab'.
- Automatisches Umbenennen des Files 'wohner.tab'.
- Anfrage, ob das alte File 'wohner.tab' gestrichen werden soll.
- Anfrage, ob angefügt werden soll.
- Anfrage, ob das bereits existierende File 'wohner.tab' umbenannt werden soll.

Welche Reaktion gibt es auf Ihrem Rechner? Welche Reaktion fänden Sie sinnvoll? Warum?

Nach dem beschriebenen Misserfolg benenne ich 'wohner.tab' um (Kommando 'ren') und wiederhole den Programmlauf. Jetzt erhalte ich das File 'wohner.tab' mit dem gewünschten Inhalt.

Nach abermaligem Umbenennen provoziere ich einen Programmabbruch während der Sortier- und Ausgabephase des Programms. Bei meinem Rechner muss ich zu diesem Zweck gleichzeitig die Tasten 'Control' und 'c' drücken (gängige Schreibweise dafür: Ctrl/C), was sich auf dem Terminal als '^C' zeigt. Bei Ihnen kann das anders sein! Der anschliessende Blick ins Fileverzeichnis lehrt, dass kein File 'wohner.tab' entstanden ist.

Hierzu sind zwei Bemerkungen erforderlich:

1. Wegen der hohen Geschwindigkeit der Rechner ist es sehr schwer, den Abbruch wirklich zu erreichen. Wenn man nach dem Doppelkreuz endlich Ctrl/C eingetippt hat, dann sind Sortieren und Ausgabe längst beendet.
   In Wirklichkeit habe ich mehrmals 50 Einträge gemacht und dann jeweils den Abbruch probiert. Nach einigen Fehlversuchen (d. h., das File 'wohner.tab' war ordnungsgemäss geschrieben worden) gelang mir das auch.
2. Die Reaktion auf den Abbruch kann durchaus anders sein. Sie hängt ab vom jeweiligen MODULA-2-System und ist oft diktiert von den Gegebenheiten des benutzten Betriebssystems.
   Es könnte sein, dass auf Ihrem Rechner ein File 'wohner.tab' entsteht, das alle oder fast alle vor dem Abbruch sortiert

geschriebenen Einträge enthält. In manchen Systemen ist das File nach dem Abbruch eine "Ruine": Man kann den Inhalt nicht lesen; nur durch Spezialkommandos des Betriebssystems kann es gelöscht oder - in seltenen Fällen - normal lesbar gemacht werden.

Die in diesem Abschnitt dokumentierten und zusätzlich empfohlenen Testläufe deuten darauf hin, dass auch sehr kurze und scheinbar voll überschaubare Programme interessante Fehler enthalten können. Ich habe die Erfahrung gemacht, dass man dann, wenn alle gutgeplanten Tests eines Programms keinen Fehler mehr ans Tageslicht befördern, einmal einen naiven "Fremdnutzer" mit dem fraglichen Programm arbeiten lassen sollte: Im allgemeinen provoziert er neue und ungeahnte Fehler!

4. Syntaxdarstellung mit EBNF und Lexik von MODULA-2

Der "Erweiterte Backus-Naur-Formalismus" (EBNF) ist ein Hilfsmittel zum Formulieren von Regeln für das Aneinanderreihen einzelner "Symbole" zu "Symbolfolgen". Er wird in diesem Buch zur Beschreibung des Aufbaus aller Sprachelemente von MODULA-2 verwendet. Oft lässt sich die Gestalt von Eingabe- und/oder Ausgabedaten eines Programms günstig in EBNF ausdrücken. Derartige Datenbeschreibungen sind verbindlicher als verbale Erklärungen.
Nach der Einführung in EBNF folgt die Darstellung der "Lexik" von MODULA-2: Aus welchen Symbolen setzen sich MODULA-2-Programme zusammen, d. h., welche Zeichenfolgen sind gültige Symbole?

4.1. Syntaxdarstellung mit EBNF

Der "Erweiterte Backus-Naur-Formalismus" (EBNF) dient zur Darstellung des Aufbaus von Symbolfolgen aus Teilfolgen. Er lässt zwei Lesarten zu:

1. Beschreibung der Erzeugung zulässiger Symbolfolgen und
2. Vorschrift zur Zulässigkeitsprüfung für Symbolfolgen.

Die Prüfung wird i. allg. als Versuch des "Nachbaus" der zu prüfenden Symbolfolge durchgeführt. Jede Vorschrift, die zulässige Symbolfolgen von unzulässigen abgrenzt, wird in der heute üblichen Terminologie der Informatik als **Syntax** bezeichnet. Syntaxdarstellungen in EBNF bestehen aus einer oder mehreren Syntaxregeln (EBNF-Regeln). Sie haben die folgende Gestalt.

Hilfssymbol = Syntaxausdruck .

Jede Regel wird mit einem Punkt abgeschlossen. Hilfssymbole sind aus Kleinbuchstaben und Unterstreichungsstrichen ("_") zusammengesetzte Zeichenketten. Ich verwende als Hilfssymbole deutsche Wörter, die etwas über die inhaltliche Bedeutung der rechts stehenden syntaktischen Konstruktion aussagen sollen. Für jedes Hilfssymbol wird genau eine Regel aufgestellt. Daher könnte man das links stehende Hilfssymbol auch als den "Regelnamen" bezeichnen. Das Gleichheitszeichen kann als "ist" oder "besteht aus" gelesen werden.

Inhaltliche Bedingungen, die sich nicht in einer Syntaxregel ausdrücken lassen, werden als verbale Anmerkungen notiert, z. B. die Forderungen, dass eine Zahl in einer bestimmten Situation kleiner als 65536 sein soll oder dass ein "Name" schon an anderer Stelle erschienen sein muss.
Zur Vermeidung von Missverständnissen verwende ich für die "eigentlichen" Symbole ab sofort den Begriff **Terminalsymbole**. Syntaxausdrücke bestehen aus Terminalsymbolen und Hilfssymbolen sowie Angaben zu Wiederholungen und Auswahlen.
Ein kleines Beispiel soll die formalere Beschreibung dieser Dinge vorbereiten. Die aus zwei EBNF-Regeln bestehende Syntax im **Bild 4.1** beschreibt ganze Zahlen in Dezimaldarstellung, die ein Vorzeichen haben dürfen.

```
zahl = ["+"|"-"] ziffer{ziffer} .
ziffer = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Verbale Interpretation:

Eine 'zahl' besteht aus einem Vorzeichen, das entfallen kann, und einer 'ziffer' sowie weiteren Ziffern, die auch entfallen können.
Eine 'ziffer' ist genau eine der Ziffern Null bis Neun.

Bild 4.1. Syntaxbeispiel: Ganze Dezimalzahlen, evtl. mit Vorzeichen

Terminalsymbole werden in Doppelapostrophe (Gänsefüsschen) eingeschlossen. Zur Verbesserung der Lesbarkeit weiche ich bei der Beschreibung der MODULA-2-Syntax ein wenig von dieser Festlegung ab und notiere die sogenannten Schlüsselwörter (z. B. MODULE, VAR, IF) ohne Gänsefüsschen. Da alle Schlüsselwörter nur aus Grossbuchstaben bestehen, ist keine Verwechslung mit Hilfssymbolen möglich.
Syntaxausdrücke bestehen aus

- Terminal- und Hilfssymbolen
- Paaren von eckigen, geschweiften und runden Klammern sowie
- senkrechten Strichen.

Jeder Syntaxausdruck ist ein Muster für den Aufbau von Terminalsymbolfolgen. Er kann als Beschreibung der Möglichkeiten zum "Aufschreiben" solcher Folgen aufgefasst werden.

Das Ableiten von Terminalsymbolfolgen aus Syntaxausdrücken ist durch folgende sieben Punkte definiert.

1. Terminalsymbole bedeuten sich selbst, d. h., sie müssen aufgeschrieben werden.
2. Ein Hilfssymbol steht für die durch die rechte Seite der zugehörigen Syntaxregel beschriebenen Folgen.
3. Wenn Konstruktionen hintereinander stehen, dann müssen auch die diesen Konstruktionen entsprechenden Terminalsymbolfolgen hintereinander geschrieben werden.
4. Der senkrechte Strich dient als Auswahloperator; lies: "oder". Jede der durch Auswahloperatoren getrennten Varianten ist möglich; genau eine muss genommen werden.
5. Konstruktionen in eckigen Klammern können weggelassen oder genau einmal verwendet werden.
6. Konstruktionen in geschweiften Klammern sind beliebig oft wiederholbar. Sie dürfen auch entfallen (0malige Wiederholung).
7. Konstruktionen in runden Klammern sind genau einmal auszuwerten.

Die Verschachtelung von Klammern ist zu beachten. Der senkrechte Strich wirkt nur innerhalb des ihn am engsten einschliessenden Klammernpaars bzw. für die gesamte rechte Seite einer Regel. In der Regel 'zahl' findet man den ersten und in der Regel 'ziffer' den zweiten Fall. Die runden Klammern (s. Punkt 7) haben nur in Zusammenhang mit dem Auswahloperator Bedeutung. Sie gestatten das "Ausklammern" gemeinsamer Teile unterschiedlicher Varianten.
Wenn wir das Zahlenbeispiel abändern und fordern, dass immer ein Vorzeichen da sein muss, dann können wir an Stelle der Regel

```
zahl = "+" ziffer{ziffer} | "-" ziffer{ziffer}
```

kürzer

```
zahl = ("+"|"-") ziffer{ziffer}
```

schreiben. Man mache sich an Hand der drei angegebenen Regeln für 'zahl' den Bedeutungsunterschied zwischen eckigen und runden Klammern in EBNF-Syntaxausdrücken klar.
In jeder EBNF-Syntax gibt es ein ausgezeichnetes "erstes" oder "oberstes" Hilfssymbol, mit dem der durch die Punkte 1 bis 7 gegebene "Aufschreibeprozess" zu beginnen ist. Es wird Satz- oder **Startsymbol** genannt. Startsymbol im Beispiel auf S. 51 ist 'zahl'.

In der mehr theoretisch orientierten Informatikliteratur werden die bisher eingeführten Dinge zu Definitionen der Begriffe "formale Grammatik" und "formale Sprache" zusammengefasst.
Eine formale Grammatik G ist ein Viertupel

G = (T, H, R, s)

mit

T Menge der Terminalsymbole (auch "Lexik" genannt)
H Menge der Hilfssymbole
R Syntax (hier Menge von EBNF-Regeln)
s Startsymbol (aus H).

Die zur Grammatik G gehörende formale Sprache L(G) ist die Menge aller gemäss Punkt 1 bis 7 aus dem Startsymbol s ableitbaren Terminalsymbolfolgen.
Bei der Definition der Bedeutung von Syntaxausdrücken wurde in Punkt 3 formuliert, dass Terminalsymbole "hintereinander" zu schreiben seien. Was heisst das? Dürfen Leerzeichen dazwischen stehen oder nicht? Dürfen vielleicht sogar Zeilenenden und irgendwelche "Kommentare" dazwischen stehen?
Das angegebene Zahlenbeispiel legt die Forderung nahe, dass die Terminalsymbole ohne Zwischenraum unmittelbar nebeneinander stehen sollten. Für Dinge wie Leerzeichen, Zeilenenden und Kommentare könnten ein Hilfssymbol 'zwischenraum' und eine zugehörige Regel eingeführt werden. Es würde dann in Syntaxausdrücken überall dort auftauchen, wo Zwischenräume erlaubt sein sollen.
Seit dem Erscheinen der Programmiersprache ALGOL-60 [Naur62] ist es üblich, formatfreie Programmnotationen zuzulassen, d. h., der Programmierer kann durch Einrückungen, Leerzeilen und Kommentare seinen Programmtext so gestalten, wie er am besten lesbar ist. Für die Syntax derartiger Sprachen - MODULA-2 gehört dazu - hat das die Konsequenz, dass jedes zweite Symbol in Syntaxausdrücken 'zwischenraum' heissen würde. Aus diesem Grunde wird die strikte Aufeinanderfolge von Terminalsymbolen nur in jenen Regeln gefordert, welche die Lexik der Sprache beschreiben. Als Terminalsymbole treten dort i. allg. nur einzelne Zeichen auf. Für alle anderen Regeln wird verbal eine "Trennzeichenregelung" formuliert. Lexik und Trennzeichenregelung bei MODULA-2 sind Gegenstand von Abschnitt 4.2.
Syntaxregeln, bei denen die unmittelbare Aufeinanderfolge der Terminalsymbole gefordert ist, treten nur im Abschnitt 4.2 dieses

Buches auf. Sie sind mit einem '&' am Zeilenanfang gekennzeichnet. Alle anderen Syntaxregeln habe ich vorn mit einem '$' oder '+' versehen. Die beiden Zeichen haben folgende Bedeutung:

$ Hier gilt die Trennzeichenregelung von MODULA-2.

\+ Wie $, aber die Regel ist unvollständig.

Aus methodischen Gründen gebe ich an einigen Stellen vorläufige Regeln an, in denen einige Alternativen fehlen. Diese mit einem + gekennzeichneten Regeln erscheinen an einer späteren Textstelle noch einmal vollständig mit einem $-Zeichen.

4.2. Lexik der Sprache

Die kleinsten eine selbständige Bedeutung tragenden Einheiten einer Programmiersprache bezeichnet man als **Morpheme**. Andere Bezeichnungen dafür sind "lexikalische Einheit" oder "Lexem". Bei der Sprache MODULA-2 gibt es sechs Klassen von Morphemen:

1. Spezialsymbole
2. Schlüsselwörter
3. Bezeichner
4. Zahlen
5. Zeichen- und Zeichenkettenkonstanten
6. Trennzeichen.

Spezialsymbole bestehen aus einem Zeichen oder aus zwei unmittelbar aufeinanderfolgenden Zeichen. In Tafel 4.1 sind alle 27 Spezialsymbole von MODULA-2 aufgeführt.

Tafel 4.1. Die Spezialsymbole von MODULA-2

Operatoren	+	-	*	/	&		:=
Relationssymbole	=	#	<>	<	>	<=	>=
Klammern(paare)	(	)	[	]	{	}	
Sonstige						^	|

MODULA-2 kennt 40 **Schlüsselwörter**, die sämtlich nur aus Grossbuchstaben bestehen. Tafel 4.2 ist eine alphabetisch geordnete Aufstellung.

Tafel 4.2. Die Schlüsselwörter von MODULA-2

AND	ELSIF	LOOP	REPEAT
ARRAY	END	MOD	RETURN
BEGIN	EXIT	MODULE	SET
BY	EXPORT	NOT	THEN
CASE	FOR	OF	TO
CONST	FROM	OR	TYPE
DEFINITION	IF	POINTER	UNTIL
DIV	IMPLEMENTATION	PROCEDURE	VAR
DO	IMPORT	QUALIFIED	WHILE
ELSE	IN	RECORD	WITH

Alle anderen Morphemklassen sind zu umfangreich, um sie durch Aufzählen ihrer Elemente beschreiben zu können. Sie enthalten potentiell unendlich viele Elemente. Man definiert deshalb mit Hilfe von EBNF-Regeln, wie die Morpheme aus einzelnen Zeichen zusammengesetzt werden.
Bezeichner bestehen aus Buchstaben und Ziffern. Sie beginnen mit einem Buchstaben und können i. allg. beliebig lang sein. Sie werden vom Programmierer als Namen für viele Dinge (z. B. Bausteine, Variablen) eingeführt.

```
&    bezeichner = buchstabe { buchstabe | ziffer }
&    buchstabe  = "a" | "b" |     | "z" | "A" | "B" |     | "Z"
&    ziffer     = "0" | "1" |     | "9"
```

Schlüsselwörter dürfen nicht als Bezeichner verwendet werden.

Beispiele (aus den Programmen im Abschnitt 2):

```
    Fragen   TextFrage   position   i   laenge   WriteLn
```

Innerhalb von Bezeichnern ist der Unterschied zwischen Gross- und Kleinbuchstaben signifikant: 'max', 'Max' und 'MAX' sind verschieden.
In der Definition der Sprache MODULA-2 werden 31 Bezeichner zur Benennung von einigen Konstanten, Prozeduren und elementaren Datentypen eingeführt. Diese Standardbezeichner sind in Tafel 4.3 zusammengestellt. Sie haben eine vordefinierte Bedeutung. Im Unter-

schied zu Schlüsselwörtern können sie vom Programmierer innerhalb von Prozeduren umdefiniert werden. Aber ich empfehle jedem Leser, es nicht zu tun.

Tafel 4.3. Die Standardbezeichner von MODULA-2

Konstanten	FALSE	NIL	TRUE		
Prozeduren	ABS	CAP	CHR	DEC	DISPOSE
	EXCL	FLOAT	HALT	HIGH	INC
	INCL	MAX	MIN	NEW	ODD
	ORD	TRUNC	VAL		
Datentypen	BITSET	BOOLEAN	CARDINAL	CHAR	INTEGER
	LONGCARD	LONGINT	LONGREAL	PROC	REAL

Zahlen werden bevorzugt als Dezimalzahlen notiert. Für ganze Zahlen können auch Oktal- und Hexadezimaldarstellungen geschrieben werden. Reelle Zahlen sind durch das Auftreten eines Dezimalpunktes charakterisiert. Der in der Skalierung erscheinende Buchstabe 'E' ist als "mal 10 hoch" zu lesen.

```
&    zahl = ganze_zahl | reelle_zahl .
&    ganze_zahl  = ziffer {ziffer}
&                | oktalziffer {oktalziffer} "B"
&                | ziffer{hexziffer} "H" .
&    oktalziffer = "0" | "1" | ... | "7"
&    hexziffer   = ziffer | "A" | "B" | ... | "F".
&    reelle_zahl = ziffer {ziffer} "." {ziffer} [skalierung]
&    skalierung  = "E" ["+"|"-"] ziffer{ziffer}
```

Abhängig vom jeweiligen Rechner und vom benutzten Compiler existieren Maximalwerte für zulässige Zahlen. Werden sie überschritten, so erscheint beim Übersetzen ein Quelltextfehler. Bei mir ist es

```
2: constant out of range.
```

Bei 8- und 16-Bit-Rechnern beträgt der Maximalwert für ganze Zahlen i. allg. 65 535, bei 32-Bit-Rechnern 4 294 976 295 (2**wl - 1, wobei 'wl' die Wortlänge und '**' der Potenzieroperator ist). Ganze Zahlen sind Konstanten der Standardtypen 'CARDINAL' oder 'INTEGER'.

Bez. der Unterscheidung s. Abschnitte 6.1 und 6.2. Reelle Zahlen gehören zum Standardtyp 'REAL', der mit einigen seiner Probleme im Abschnitt 11 erläutert wird.
Viele Dienstprogramme, die Angaben über rechnerinterne Speicherinhalte ausschreiben, liefern ihre Ergebnisse in Oktal- oder Hexadezimaldarstellung. Wenn es Anlass gibt, solche Werte als Konstanten in einem MODULA-2-Programm zu benutzen, dann ist das Umrechnen in die Dezimalnotation lästig und fehleranfällig. Wohl aus diesem Grund dürfen ganze Zahlen auch in Oktal- bzw. Hexadezimalnotation geschrieben werden.
Bei Oktalzahlen ist die Darstellungsbasis 8. Bei Hexadezimalzahlen ist es 16. Die Hexadezimalziffern "A" bis "F" stehen für die Werte 10 bis 15. Der Wert W einer n-ziffrigen Zahldarstellung zur Basis b ergibt sich als

```
W = v(z1) + v(z2)*b**1 + v(z3)*b**2 + ... + v(zn)*b**(n-1),
```

wobei z1 bis zn die von rechts beginnend durchnumerierten Ziffern sind und die Funktion v(x) den Wert von Ziffer x angibt. Es ist also v(7)=7 und v(D)=13.
Als Beispiel berechne ich die Werte der drei MODULA-2-Zahlen 237, 355B und 0EDH nach dieser Formel.

```
v(7) + v(3)*10**1 + v(2)*10**2 = 7 + 3*10 + 2*100 = 237
v(5) + v(5)*8**1 + v(3)*8**2 = 5 + 5*8 + 3*64 = 237
v(D) + v(E)*16**1 + v(0)*16**2 = 13 + 14*16 + 0*256 = 237
```

Bitte beachten Sie, dass eine hexadezimal dargestellte Zahl in MODULA-2 immer mit einer 'ziffer' beginnt! Wenn die erste Ziffer eine der Hexadezimalziffern "A" bis "F" ist, dann muss eine Null davor geschrieben werden, z. B. 0EDH. Diese Festlegung wurde getroffen, damit der Compiler die 'ganze_zahl' 0EDH vom 'bezeichner' EDH unterscheiden kann.

Zeichenketten sind beliebige Folgen von Zeichen, die entweder in Apostrophe oder in Anführungsstriche (Gänsefüsschen) eingeschlossen sind. Zeichenketten werden oft auch als "Strings" (engl.) bezeichnet.
Innerhalb von Zeichenketten sind Zeilenwechsel und das zur Einschliessung verwendete Zeichen verboten. Es ist demzufolge unmöglich, eine Zeichenkette zu notieren, die sowohl einen Apostroph als auch ein Gänsefüsschen enthält.

Die dritte in der Syntaxregel angegebene Variante beschreibt ein einzelnes Zeichen, und zwar jenes Zeichen, das im ASCII-Zeichensatz (s. Tafel 4.4 auf der nächsten Seite) die Nummer hat, die vor dem "C" in Oktaldarstellung notiert ist.

```
&    zeichenkette = "'" {zeichen_ausser_apostroph} "'"
&                 | """ {zeichen_ausser_doppelapostroph} """
&                 | oktalziffer {oktalziffer} "C"
```

Der erste Grossbuchstabe des deutschen Alphabets kann auf drei Arten als MODULA-2-Konstante notiert werden:

'A' oder "A" oder auch 101C.

Die Länge einer Zeichenkette ist die Anzahl der Zeichen zwischen den Einschliessungszeichen. Zeichenketten der Länge 1 sind auch Konstanten des Standardtyps CHAR. In "älteren" MODULA-2-Systemen wurden Zeichenkettenkonstanten der Länge 1 nur als CHAR-Konstanten betrachtet.

Es gibt vier Sorten von Trennzeichen: das Leerzeichen, den horizontalen Tabulator, den Zeilenwechsel und Kommentare. Kommentare sind beliebige Zeichenfolgen, die mit "(*" und "*)" eingeklammert sind. Sie können ineinandergeschachtelt werden.

```
&    kommentar = "(*" { textstueck | kommentar } "*)"
```

Ein 'textstueck' ist eine Zeichenfolge, in der die Zeichenkombinationen "*)" und "(*" nicht auftreten.

Es folgen zwei korrekt gebildete Kommentare.

```
(* Das ist ein Kommentar *)
(* Das (*ist*) ein (* (*arg*) verschachtelter*)
   zweizeiliger Kommentar *)
```

Die nächsten zwei Beispiele zeigen beliebte Fehler.

```
(* Das ist kein *) Kommentar *)
(* Hier ist der (* Wurm drin *)
```

Grösste Verwunderung löst vor allem der zweite Fall aus: Der Compiler wird den gesamten Quelltext hinter diesem "Kommentar" als zum Kommentar gehörend auffassen, da er nach einer schliessenden Kommentarklammer sucht, die zur ersten öffnenden gehört. Kommentare haben i. allg. keine inhaltliche Bedeutung beim Übersetzen und

Tafel 4.4. ASCII-Zeichensatz

Jeder Eintrag enthält den Zeichencode in Dezimal-, Hexadezimal- und Oktaldarstellung sowie das entsprechende Zeichen.
Ausgewählte Steuerzeichen sind in Tafel 4.5 kurz erläutert.

0	00	000	nul	32	20	040		64	40	100	@	96	60	140	
1	01	001	soh	33	21	041	!	65	41	101	A	97	61	141	a
2	02	002	stx	34	22	042	"	66	42	102	B	98	62	142	b
3	03	003	etx	35	23	043	#	67	43	103	C	99	63	143	c
4	04	004	eot	36	24	044	$	68	44	104	D	100	64	144	d
5	05	005	enq	37	25	045	%	69	45	105	E	101	65	145	e
6	06	006	ack	38	26	046	&	70	46	106	F	102	66	146	f
7	07	007	bel	39	27	047	'	71	47	107	G	103	67	147	g
8	08	010	bs	40	28	050	(	72	48	110	H	104	68	150	h
9	09	011	ht	41	29	051	)	73	49	111	I	105	69	151	i
10	0A	012	lf	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	0B	013	vt	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	0C	014	ff	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	0D	015	cr	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	0E	016	so	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	0F	017	si	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	dle	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	dc1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	dc2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	dc3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	dc4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	nak	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	syn	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	etb	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	can	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	em	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	sub	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	esc	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	fs	60	3C	074	<	92	5C	134	\	124	7C	174	|
29	1D	035	gs	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	rs	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	us	63	3F	077	?	95	5F	137	_	127	7F	177	del

Tafel 4.5. Bedeutung einiger ASCII-Steuerzeichen

nul	bei MODULA-2 internes Endekennzeichen von Zeichenketten
bel	akustisches Signal (engl. bell)
bs	Rückwärtsschritt (engl. backspace)
ht	Horizontaltabulator
lf	Zeilenvorschub (engl. line feed)
ff	Seitenvorschub (engl. form feed)
cr	Wagenrücklauf (engl. carriage return)
esc	ESCAPE-Zeichen (leitet Steuerzeichenfolgen für Drucker- und Bildschirmausgaben ein)
40C	Leerzeichen
del	Löschen (engl. delete)

Abarbeiten von Programmen; sie sollen für den Programmierer und seine Nachfolger die Lesbarkeit des Programms verbessern.
Die einzige Ausnahme bilden sogenannte Compileroptionen zur Übermittlung von Instruktionen an den Compiler. Sie haben die Gestalt eines Kommentars, der mit einem speziellen Zeichen (normalerweise "$") beginnt. Im Normalfall benötigt man diese Compileroptionen nicht. Genauere Angaben über solche Dinge und ihre Verwendung sind stark systemabhängig und sollten der jeweiligen Compilerdokumentation entnommen werden.
Die am Ende des vorigen Abschnitts erwähnte Trennzeichenregelung für die Notation von MODULA-2-Programmen lässt sich in drei Punkten formulieren:

1. Trennzeichen dürfen zwischen je zwei Vertretern aus der Menge der Spezialsymbole, Schlüsselwörter, Bezeichner, Zahlen sowie Zeichen- und Zeichenkettenkonstanten stehen.
2. Wenn zwei Schlüsselwörter oder ein Schlüsselwort und ein Bezeichner nebeneinander stehen, dann muss dazwischen ein Trennzeichen geschrieben werden.
3. Wo ein Trennzeichen erlaubt oder gefordert ist, dort dürfen beliebig viele Trennzeichen auftreten.

Diese Festlegungen machen es möglich, die Morphemklasse 'trennzeichen' in allen weiteren Syntaxregeln wegzulassen.

5. Blöcke, Objektvereinbarungen und Gültigkeitsbereiche

Die fundamentale Programmstruktur bei MODULA-2 ist der aus einem Vereinbarungsteil und einem Aktionsteil bestehende Block. Der Aktionsteil beschreibt einen Informationsverarbeitungsprozess. Im Vereinbarungsteil werden die passiven und aktiven Objekte dieses Prozesses eingeführt und benannt.
Innerhalb eines Blockes können Unterprogramme vereinbart werden. Ihr "Körper" ist wiederum ein Block. Auch diese Blöcke dürfen Unterprogramme vereinbaren usw. Bei MODULA-2 sind beliebig tiefe Verschachtelungen von Blöcken möglich.
Die damit aufgeworfenen Fragen nach Gültigkeitsbereich, Sichtbarkeit und Eindeutigkeit der in den Vereinbarungsteilen verschachtelter Blöcke eingeführten Namen (Bezeichner) sind Gegenstand von Abschnitt 5.4.
Damit jeder Leser die angegebenen MODULA-2-Beispiele auch wirklich ausprobieren und ein wenig experimentieren kann, beginne ich mit der Beschreibung des Aufbaus von Programmoduln und Importklauseln.

5.1. Programmodule, Importe und Blöcke

Das Startsymbol der MODULA-2-Grammatik ist 'uebersetzungseinheit'. Die folgende Syntaxregel wird im Abschnitt 10.1 vervollständigt.

```
+    uebersetzungseinheit = programmodul .
$    programmodul = MODULE bezeichner ";"
$                   {import ";"} block bezeichner "."
```

Hinter MODULE und nach dem 'block' muss derselbe 'bezeichner' notiert werden.

Die Übersetzungseinheit Programmodul beginnt mit dem Schlüsselwort MODULE, dem ein Bezeichner folgt. Der Bezeichner spielt die Rolle eines Programmnamens. Er muss vor dem den Programmodul beendenden Punkt wiederholt werden. Diese Namenswiederholung ist typisch für MODULA-2 und wird auch bei Prozeduren gefordert. Sie erzwingt eine bessere Lesbarkeit von Programmtexten und wird anfangs - vor allem

von PASCAL-Programmierern - häufig vergessen. Die einschlägigen Fehlermeldungen des Compilers lauten bei mir:

20: identifier expected bzw.

24: block name at the END does not match.

Im ersten Fall wurde der Name vergessen. Im zweite Fall wurde er falsch geschrieben. Der Programmname ist für die Abarbeitung des Programms bedeutungslos; aber es gibt Konventionen zur Ableitung der Filenamen von Quelltextfiles aus den Namen der darin gespeicherten Übersetzungseinheiten, deren Einhaltung gewisse technologische Vorteile bringt (s. Abschn. 10.3).

Importe bereiten die Bezugnahme auf separat übersetzte Programmbausteine vor. Sogar der Programmierer, der zunächst noch keine privaten Bausteine schreiben will, muss über das Importkonzept Bescheid wissen; denn sämtliche Eingabe- und Ausgabeoperationen erfordern den Rückgriff auf vorgefertigte Bausteine.

```
$    import = IMPORT liste_von_modulnamen
$           | FROM modulname IMPORT bezeichnerliste .
```

Die 'bezeichnerliste' darf nur solche Namen enthalten, die vom Modul 'modulname' exportiert werden.

```
$    liste_von_modulnamen = bezeichnerliste .
$    modulname = bezeichner
```

Durch einen 'import' der ersten Form werden alle Bausteinnamen, die in der 'liste_von_modulnamen' stehen, im Programmodul "bekanntgemacht", und man importiert sämtliche Bezeichner, die von den aufgezählten Bausteinen exportiert werden. Diese Bezeichner können in dem zugehörigen 'block' nicht direkt benutzt werden. Ihnen sind der entsprechende Bausteinname und ein Punkt voranzustellen. Man spricht von "Bezeichnern, die durch den Modulnamen qualifiziert" werden, oder kurz von "qualifizierten Namen".

```
+    q_name = [modulname "."] bezeichner
```

Das Hilfssymbol 'q_name' wird immer dort auftauchen, wo neben direkt im Vereinbarungsteil eingeführten Bezeichnern auch importierte Namen möglich sind. Die Syntaxregel 'q_name' wird im Abschnitt 13.1 in Zusammenhang mit "lokalen Moduln" vervollständigt.

Ein 'import' der zweiten Form macht die in der 'bezeichnerliste' genannten Namen im Programmodul bekannt: Sie können im zugehörigen 'block' benutzt werden. Verglichen mit der ersten Importvariante spart man im Programmtext evtl. Schreibarbeit: Der Modulname erscheint nur einmal in der FROM-IMPORT-Klausel und nicht bei jeder Bezugnahme auf ein importiertes Objekt als Bestandteil des qualifizierten Namens.
In den bisherigen Beispielen habe ich die zweite Form des Imports bevorzugt. Programm 3.1 könnte umgeschrieben werden.

```
MODULE Hier;
IMPORT InOut;
BEGIN
  InOut.WriteString("Hier bin ich!"); InOut.WriteLn;
END Hier.
```

Auch eine Mixtur beider Formen ist erlaubt. Zu den Importklauseln

```
IMPORT InOut;
FROM InOut IMPORT WriteString;
```

würden die Bezugnahmen 'WriteString' (importierter Bezeichner) und 'InOut.WriteLn' (qualifzierter Name) gehören.
Warum gibt es mehrere Möglichkeiten, dieselbe Sache auszudrücken? FROM-IMPORT-Klauseln führen zu kürzeren Programmen. Aber sie bergen bei unabhängig voneinander hergestellten Programmbausteinen die Gefahr von "Namenskollisionen" in sich. Der Import von Modulnamen vermeidet Namenskollisionen. Und der Umstand, dass man jedem qualifizierten Namen ansieht, woher der fragliche Bezeichner importiert ist, verbessert die Lesbarkeit von Programmen ungemein.
Betrachten wir ein Beispiel! Wenn in einem Programm die Notwendigkeit besteht, von mehreren Bausteinen zu importieren, dann kann es sein, dass zwei Bausteine (etwa 'B1' und 'B2') denselben Bezeichner (z. B. 'x') exportieren und beide 'x' gebraucht werden. Was tun? Die Variante

```
FROM B1 IMPORT x;
FROM B2 IMPORT x;
```

wird mit der Fehlernachricht

```
70: identifier specified twice in importlist
```

zurückgewiesen. Das Importieren der beiden Modulnamen durch

```
IMPORT B1, B2;
```

und Bezugnahmen der Form 'B1.x' bzw. 'B2.x' leisten das Verlangte.

Die fundamentale Programmstruktur bei MODULA-2 ist der **Block**. Ein Block besteht aus zwei Teilen: Zuerst werden sowohl die "passiven" Objekte (Daten) als auch die abarbeitungsfähigen Objekte (Prozeduren) vereinbart; danach kommt die mit diesen Objekten zu vollführende Aktionen- oder Anweisungsfolge.

```
$    block = {vereinbarung} [BEGIN anweisungsfolge] END .
+    vereinbarung = VAR {variablenvereinbarung ";"}
+                 | CONST {konstantenvereinbarung ";"}
+                 | prozedurvereinbarung ";"
$    anweisungsfolge = anweisung {";" anweisung}
```

Objekte, wie sie in der Sprache MODULA-2 verfügbar sind, können abstrakt als Paare aus einem Namen und einem Wert verstanden werden. Der Name dient zur Bezugnahme auf den Wert.

> Jedes Objekt, mit dem in der zum Block gehörenden Anweisungsfolge Aktionen ausgeführt werden sollen, muss entweder vereinbart werden, oder sein Name muss importiert werden. Die importierten Bezeichner und die Namen aller in den Vereinbarungen eines Blockes eingeführten Objekte müssen voneinander verschieden sein.

Es gibt in MODULA-2 auch Objekte, deren Name kein 'bezeichner' ist. Das betrifft einerseits die Literale und andererseits die dynamischen Variablen.

Literale sind die in den Regeln 'zahl' und 'zeichenkette' definierten Objekte. Sie werden durch ihre "Gestalt" (die Ziffernfolge bzw. die geklammerte Zeichenfolge) identifiziert. Beispielsweise wird durch die Notation

```
13
```

auf ein Objekt Bezug genommen, dessen Wert die Zahl 13 ist. Man bezeichnet die Literale auch als unbenannte Konstanten.

Dynamische Variablen können nur über sogenannte Zeiger oder Pointer erreicht werden. Pointertypen, dynamische Variablen und Beispiele für damit konstruierbare dynamische Datenstrukturen sind Gegenstand von Abschnitt 12.

5.2. Variablen-, Konstanten- und Prozedurvereinbarungen

MODULA-2 kennt drei Varianten zur Vereinbarung von Objekten. Variablen sind Objekte, deren Wert während der Abarbeitung des Programms verändert werden kann.

```
$    variablenvereinbarung = bezeichnerliste ":" typ .
$    bezeichnerliste = bezeichner {"," bezeichner}
+    typ = q_name .
```

Eine 'variablenvereinbarung' führt die in der 'bezeichnerliste' aufgezählten Variablennamen ein und legt fest, dass diese Variablen nur Werte aus dem durch 'typ' beschriebenen Wertebereich annehmen können.
Beispiel für die Vereinbarung von Variablen (aus Programm 2.2):

```
VAR
  position, schluss, maxNr, maxLeute, i: INTEGER;
```

Vorerst lasse ich für 'typ' nur die zehn in Tafel 4.3 angegebenen Bezeichner der Standardtypen und importierte Typnamen zu. Die Standardtypen sind Gegenstand des Abschnitts 6. Das vollständige Typkonzept von MODULA-2 wird im Abschnitt 9 vorgestellt.
Variablen haben keinen definierten Anfangswert. Ein besonders in Anfängerprogrammen häufig beobachtbarer Fehler ist die Bezugnahme auf den Wert einer Variablen, noch bevor ihr erstmals ein Wert zugewiesen wurde.
Die beiden anderen Formen der Objektvereinbarung betreffen Objekte mit unveränderlichem Wert. Jede 'konstantenvereinbarung' und jede 'prozedurvereinbarung' führen genau einen Objektnamen ein und verbinden ihn mit einem festen Wert.
Eine benannte Konstante hat einen Wert, der vom Compiler durch Berechnung eines Konstantenausdrucks ermittelt wird.

```
$    konstantenvereinbarung = bezeichner "=" konstantenausdruck .
+    konstantenausdruck = zahl | zeichenkette | q_name .
   'q_name' muss eine Bezugnahme auf eine benannte Konstante sein.
```

Konstantenausdrücke beschreiben arithmetische und logische Verknüp-

fungen von solchen Operanden, die ihrerseits Konstanten sind. Wegen der grossen Ähnlichkeit zu allgemeinen Ausdrücken, erläutere ich sie im Abschnitt 7 mit und beschränke mich in der angegebenen vorläufigen Syntaxregel auf die elementaren Formen, in denen keine Operationen vorkommen.
Beispiel für die Vereinbarung einer Konstante (aus Programm 2.2):

```
CONST
  Kapazitaet = 70;
```

Die dritte Variante zur Vereinbarung von MODULA-2-Objekten betrifft Prozeduren (auch "Unterprogramme" oder "Subroutinen" genannt). Eine Prozedur ist die Beschreibung einer mit einem Namen versehenen Aktionenfolge. Die Aktionenfolge kann durch "Nennung" des Prozedurnamens aktiviert werden. Syntaktisch ist eine Prozedur ein 'block' mit einem 'prozedurkopf'.

```
$   prozedurvereinbarung = prozedurkopf ";" block bezeichner
$   prozedurkopf = PROCEDURE bezeichner signatur .
```

Der 'bezeichner' nach 'block' muss mit dem im 'prozedurkopf' übereinstimmen (Prozedurname).

Der zum Prozedurnamen gehörende Wert ist der hinter 'prozedurkopf' und Semikolon folgende 'block'. Man bezeichnet ihn auch als "Prozedurkörper". Diese erweiterte Auffassung des Begriffs "Wert" ergibt sich aus der Tatsache, dass MODULA-2 sogenannte Prozedurtypen zulässt; man kann Variablen vereinbaren, deren Wert eine Prozedur ist. Als Beispiele für Prozedurvereinbarungen verweise ich auf 'TextFrage' und 'ZahlFrage' im Programm 2.4.
Die 'signatur' fixiert die bei der Benutzung einer Prozedur verbindliche, d. h. vom Compiler überprüfbare Schnittstelle für den Datenaustausch mit dem Prozedurkörper.
Wie in vielen anderen Sprachen gibt es bei MODULA-2 zwei Sorten von Prozeduren:

1. Funktionsprozeduren (oder kurz: Funktionen) liefern einen Resultatwert. Sie werden innerhalb von "formelmässig notierten Berechnungen" (Ausdrücke, s. Abschn. 7) durch sogenannte Funktionsaufrufe aktiviert.
   Der von der Funktionsprozedur zurückgegebene Wert geht auf der Position des Funktionsaufrufs in die Ausdrucksberechnung ein.

2. "Reine" Prozeduren geben keinen Resultatwert zurück. Sie werden mittels einer Prozeduranweisung (auch 'prozeduraufruf' genannt) aktiviert.

Die 'signatur' enthält neben den Parametern auch die Festlegung, ob es sich um eine reine oder um eine Funktionsprozedur handelt.

```
$    signatur = [ "(" [f_par_liste] ")" [":" typname] ]
   Der 'typname' darf keinen ARRAY-, RECORD- oder
   Prozedurtyp beschreiben.
$    f_par_liste = f_par_abschnitt {";" f_par_abschnitt}
$    f_par_abschnitt = [VAR] bezeichnerliste ":" formaler_typ .
$    formaler_typ = [ARRAY OF] typname .
$    typname = q_name .
```

Wenn am Ende der 'signatur' ein Doppelpunkt und ein Typname stehen, dann handelt es sich um eine Funktionsprozedur, andernfalls um eine reine Prozedur. Der Typname gibt den Typ des von der Funktion als Resultat gelieferten Wertes an.
Die runden Klammern schliessen eine Liste mit Vereinbarungen sogenannter formaler Parameter ein. **Formale Parameter** sind Namen, die innerhalb der Prozedur wie Variablenbezeichner verwendet werden können. Ein 'f_par_abschnitt' führt die in der 'bezeichnerliste' aufgezählten Namen als formale Parameter des Typs 'formaler_typ' ein. Die ARRAY-OF-Konstruktion in der Syntaxregel 'formaler_typ' erläutere ich im Abschnitt 9.2.2 nach Einführung der ARRAY-Typen.
Die Signatur parameterloser Funktionsprozeduren besteht aus öffnender und schliessender runder Klammer, Doppelpunkt und Resultattyp. Parameterlose reine Prozeduren haben eine leere Signatur.
Wer die Syntaxregeln kritisch betrachtet, der wird feststellen, dass auch die nur aus den beiden runden Klammern bestehende Signatur möglich ist. Und die kann doch wohl nichts anderes als eine parameterlose reine Prozedure bedeuten? Ja! Das ist kein Fehler des Autors, sondern eine Unschönheit der MODULA-2-Sprachdefinition. Ich rate Ihnen, von dieser zweiten Variante keinen Gebrauch zu machen.

Bei jedem Funktions- bzw. Prozeduraufruf werden für die formalen Parameter aktuelle Parameter eingesetzt (Parametervermittlung). Was der Terminus "eingesetzt" bedeutet und wo sich das Objekt befindet,

auf das mittels eines formalen Parameters Bezug genommen wird, das hängt von der Parameterart ab.
MODULA-2 kennt zwei Parameterarten. Hinter ihnen verbergen sich zwei wesentlich verschiedene Mechanismen der Parametervermittlung. Syntaktisch unterscheiden sich die beiden Parameterarten durch das Auftreten bzw. Nichtauftreten des Schlüsselworts VAR im jeweiligen 'f_par_abschnitt'.

1. Referenzparameter (mit VAR) dienen dem Zugriff auf Objekte, die beim Funktions- bzw. Prozeduraufruf als aktuelle Parameter angegeben werden und ausserhalb der Prozedur liegen: Vor der Abarbeitung des Prozedurkörpers wird das betreffende Objekt ermittelt und als Parameterobjekt fixiert. Jede Bezugnahme auf den formalen Parameter ist ein Zugriff auf das zum Zeitpunkt des Aufrufs ermittelte Objekt.
2. Wertparameter (ohne VAR) sind "private" Objekte einer Prozedur, die bei jeder Prozeduraktivierung automatisch mit einem Anfangswert belegt werden: Als Bestandteil des Prozedur- oder Funktionsaufrufs wird vor der Abarbeitung des Prozedurkörpers der Wert eines als aktueller Parameter übergebenen Ausdrucks berechnet und dem Parameterobjekt zugewiesen. Alle Bezugnahmen auf den formalen Parameter sind Zugriffe zu diesem "privaten" Parameterobjekt.

Die für den Einsatz dieser oder jener Parameterart wichtige Quintessenz aus dem Gesagten ist:

1. Mit Wertparametern können nur Daten aus der Umgebung des Aufrufs an die Prozedur übermittelt werden.
2. Sollen auch Werte aus der Prozedur an die Aufrufumgebung fliessen, dann muss man Referenzparameter benutzen, die den Datenaustausch in beiden Richtungen gestatten.

Als instruktives Beispiel zum Unterschied der Parameterarten gebe ich drei Varianten einer Prozedurvereinbarung an. Die zu lösende Aufgabe lautet:

> **Vertauschen der Werte von zwei INTEGER-Variablen, die als Parameter übergeben werden.**

Welche Prozedur leistet das Verlangte? Das Innenleben aller drei Prozeduren ist gleich. Unterschiede gibt es nur bei den formalen Parametern. Deshalb gebe ich bei 'tausch2' und 'tausch3' nur die Prozedurköpfe an. Der Wertzuweisungsoperator ':=' macht den Wert

der rechten Seite zum Wert des links notierten Objekts (s. Abschnitt 8.1).

```
PROCEDURE tausch1(v1: INTEGER; VAR v2: INTEGER);
VAR  h: INTEGER;
BEGIN
  h:=v1; v1:=v2; v2:=h;
END tausch1;

PROCEDURE tausch2(VAR v1: INTEGER; v2: INTEGER);

PROCEDURE tausch3(VAR v1: INTEGER; VAR v2: INTEGER);
```

Bemerkung: Die Parameterangaben im Prozedurkopf 'tausch3' könnten auch gemeinsam in einem einzigen 'f_par_abschnitt' notiert werden:

```
PROCEDURE tausch3(VAR v1,v2: INTEGER);
```

Nur 'tausch3' leistet das Verlangte. In 'tausch1' dringt die Wirkung von 'v1:=v2;' nicht nach ausserhalb der Prozedur. Bei 'tausch2' tritt bez. der Anweisung 'v2:=h;' dasselbe auf. Nur Zugriffe auf Referenzparameter wirken immer auf das als aktueller Parameter angegebene Objekt.

5.3. Abstrakte Signatur von Prozeduren

Die als formale Parameter einer Prozedur auftretenden Bezeichner spielen innerhalb der Prozedur eine wichtige Rolle: Sie werden im Prozedurkörper für Bezugnahmen auf die tatsächlichen (aktuellen) Parameter benötigt.
Die bei der Benutzung einer Prozedur (Funktions- oder Prozeduraufruf) erforderliche Parametervermittlung basiert in MODULA-2 auf Reihenfolge, Art und Datentyp der Parameter. Wie die formalen Parameter heissen ist dabei völlig uninteressant.
Diese Feststellung motiviert den Begriff "abstrakte Signatur" als Charakterisierung gleichartiger Prozedurköpfe. "Gleichartig" bedeutet, dass von den als formale Parameter benutzten Namen abstrahiert wird.
Abstrakte Signaturen bilden im Abschnitt 9.4 die wesentliche Basis

für die Einführung von Prozedurtypen. Dort wird das im folgenden definierte Hilfssymbol 'abstrakte_signatur' verwendet.

```
$    abstrakte_signatur = [ "(" [f_typ_liste] ")" [":" typname] ]
   Der 'typname' darf sich nicht auf einen ARRAY-, RECORD- oder
   Prozedurtyp beziehen.
$    f_typ_liste = f_typ_angabe {"," f_typ_angabe}
$    f_typ_angabe = [VAR] formaler_typ .
```

Eine 'f_typ_angabe' mit VAR beschreibt einen Referenzparameter. Fehlt das VAR, dann handelt es sich um einen Wertparameter. Die als Beispiele zum Unterschied zwischen Referenz- und Wertparametern angegebenen Prozeduren haben folgende abstrakte Signaturen:

```
tausch1  ---  (INTEGER, VAR INTEGER)
tausch2  ---  (VAR INTEGER, INTEGER)
tausch3  ---  (VAR INTEGER, VAR INTEGER).
```

Eine Zusammenfassung aufeinanderfolgender gleichartiger Parameter zu Gruppen ist nicht möglich: Jeder Parameter muss einzeln notiert werden.

5.4. Gültigkeitsbereiche und Lebensdauer von Objekten

MODULA-2-Blöcke sind rekursiv definiert, sie können beliebig tief verschachtelt werden; denn Prozedurvereinbarungen enthalten einen 'block' und sind als spezielle Form von 'vereinbarung' Bestandteil eines Blockes (s. Syntaxregeln im Abschn. 5.2).
Das rekursive Blockkonzept wirft bez. Eindeutigkeit und Sichtbarkeit von Bezeichnern mehrere Fragen auf:

1. Wie ist die Forderung zu interpretieren, dass alle importierten und in den Vereinbarungen eines Blockes eingeführten Namen verschieden sein müssen?
2. Welche Bezeichner dürfen innerhalb einer Prozedur verwendet werden?

Diese Fragen betreffen das Problem des Gültigkeitsbereichs von Vereinbarungen und Bezeichnern.
Jede direkt im Vereinbarungsteil eines Blocks auftretende 'vereinbarung' heisst lokal zu diesem Block. Der Zusatz "direkt" soll

solche Vereinbarungen ausschliessen, die zu innerhalb des betrachteten Blocks liegenden Blöcken gehören. Für jede Vereinbarung gibt es also genau einen Block, zu dem sie lokal ist.
Der Begriff "lokal" wird auf die in einer 'vereinbarung' eingeführten (Variablen-, Konstanten- und Prozedur-)Bezeichner übertragen, und ausserdem werden alle formalen Parameter einer Prozedur als lokal zum Prozedurkörper betrachtet.
Damit ist die **Menge der lokalen Bezeichner eines Blocks** eindeutig definiert.
Die Menge der lokalen Namen des Prozedurblocks 'ZahlFrage' im Programm 2.4 besteht aus den Bezeichnern

'anforderung', 'untereGrenze', 'obereGrenze', 'zahl' und 'i'.

Betrachten wir als erstes die Eindeutigkeit der Bezeichner eines Blockes.
In MODULA-2 wird gefordert, dass die

lokalen Bezeichner eines Blockes voneinander verschieden

sein müssen.
Wenn der Block ein Modulblock mit Importen ist, dann gibt es die Zusatzforderung, dass die

importierten Bezeichner sich untereinander und von allen lokalen Bezeichnern unterscheiden

müssen.
Die Fehlermeldungen, mit denen Verstösse gegen diese Regelung von meinem Compiler abgestraft werden, lauten

72: identifier declared twice und
70: identifier specified twice in import list.

Der Gültigkeitsbereich einer Vereinbarung ist jener Teil eines Programmtextes, in welchem die durch die Vereinbarung eingeführten Bezeichner mit der in der Vereinbarung fixierten Bedeutung verwendet werden dürfen.
Es hat sich eingebürgert, einfach vom Gültigkeitsbereich eines Bezeichners zu sprechen. Man kann das tun, aber man muss sich darüber im klaren sein, dass hier mehr gemeint ist als nur der Bezeichner: Es geht immer um den Bezeichner mit der ihm in einer ganz konkreten Vereinbarung zugeschriebenen Bedeutung (z. B. um den Bezeichner 'i' als Name einer INTEGER-Variablen).
In diesem Sinne wird der Gültigkeitsbereich eines vereinbarten oder importierten Bezeichners durch zwei Regeln festgelegt. In der

ersten Regel geht es lediglich um die Blockverschachtelung; die zweite Regel formuliert eine einschränkende Reihenfolgebedingung.

1. Der Gültigkeitsbereich eines Bezeichners erstreckt sich über den gesamten Block, zu dem er lokal ist bzw. in dem er importiert wird; ausgenommen sind "lokale Module" (s. Abschn. 13.1) und solche Prozedurblöcke, in denen er erneut als lokaler Bezeichner auftritt.
2. Wenn ein Bezeichner, der in einer Vereinbarung V1 eingeführt wird, in einer Vereinbarung V2 verwendet wird und diese Verwendung nicht in einem Aktionsteil ('anweisungsfolge' in 'block') liegt, dann muss V1 textlich vor V2 stehen.

Ergänzungen zu diesen Regeln sind im Zusammenhang mit RECORD- und Pointer-Typen (s. Abschnitte 9.2.1 und 12) erforderlich.

Die wichtigste Folgerung aus Regel 1 ist die folgende:

> Wenn ein Modul oder eine Prozedur 'A' eine Prozedur 'B' enthält, dann ist es erlaubt, sowohl in 'A' als auch in 'B' denselben Bezeichner (z. B. 'n') einzuführen; allerdings bedeutet 'n' innerhalb von 'B' etwas anderes als in 'A'.

Man sagt auch, das in 'A' eingeführte 'n' sei in 'B' nicht "sichtbar". Die Vereinbarung in 'B' "überdeckt" die aus 'A'.

Im Programmodul 'Regel1' (s. Programm 5.1) ist der Bezeichner 'n' dreimal vereinbart. Wenn Sie dieses Programm übersetzen und abarbeiten, dann erscheinen auf Ihrem Bildschirm die Zahlen

```
2  4  2  1
```

in einer Zeile. Zur Erläuterung dieses Ergebnisses sind zwei Fragen zu klären:

1. Welche Vereinbarungen gehören zu den Verwendungen des Bezeichners 'n' in den Zeilen 14, 18, 23 und 27?
2. In welcher Reihenfolge werden die vier 'WriteInt'-Aufrufe abgearbeitet?

Die Vereinbarung von Zeile 10 gilt im gesamten Modulblock von 'Regel1'; ausgenommen sind aber die Prozedurblöcke von 'p' und 'r', weil 'n' dort jeweils erneut vereinbart wird. Diese Vereinbarung ist also letztlich nur für die Verwendung des Bezeichners 'n' auf Zeile 27 zuständig. Die auf Zeile 16 stehende Vereinbarung gilt im gesamten Prozedurblock von 'p', auch innerhalb von 'q'. 'n' auf den Zeilen 14 und 18 ist also eine Konstante mit dem Wert 2. Zum 'n' auf Zeile 23 gehört die Vereinbarung 'n = 4;' von Zeile 21.

```
 1 MODULE Regel1;
 2 (*--------------------------
 3   Beispiel:
 4   - Gueltigkeit von
 5     Bezeichnern,
 6   - Mehrfachvereinbarung
 7 ---------------------------*)
 8 IMPORT InOut;
 9 (*--------------------------*)
10 CONST n = 1;
11 PROCEDURE p;
12   PROCEDURE q;
13     BEGIN
14       r; InOut.WriteInt(n,3);
15     END q;
16   CONST n = 2;
17   BEGIN
18     InOut.WriteInt(n,3); q;
19   END p;
20 PROCEDURE r;
21   CONST n = 4;
22   BEGIN
23     InOut.WriteInt(n,3);
24   END r;
25 (*--------------------------*)
26 BEGIN
27   p; InOut.WriteInt(n,3);
28 END Regel1.
```

```
 1 MODULE Regel2;
 2 (*-----------------------
 3   Beispiel:
 4   - Gueltigkeit von
 5     Bezeichnern,
 6   - Reihenfolgebedingung
 7 ------------------------*)
 8 IMPORT InOut;
 9 (*----------------------*)
10 CONST
11   M1 = M;
*****       ^ 73
12 PROCEDURE s;
13 BEGIN
14   InOut.WriteInt(M,3);
15 END s;
16 CONST
17   M = 12;
18   M2 = M;
19 (*----------------------*)
20 BEGIN
21 END Regel2.

Erläuterung der Fehlernummer
bei meinem Compiler:
  73: identifier
      not declared
```

Programm 5.1. Gültigkeitsbereich von Bezeichnern
links: Mehrfachbenutzung desselben Namens
rechts: Reihenfolgebedingung bei Vereinbarungen

Die Abarbeitung beginnt in Zeile 27 mit dem Aufruf von 'p'. Wenn der Anweisungsteil einer aufgerufenen Prozedur vollständig abgearbeitet ist, dann wird mit der unmittelbar hinter ihrem Aufruf notierten Anweisung fortgesetzt. Der Reihe nach werden die 'WriteInt'-Aufrufe von den Zeilen 18, 23, 14, 27 ausgeführt.

Die rechte Seite von Programm 5.1 soll Regel 2 illustrieren. Der Bezeichner 'M' ist lokal zum Modulblock 'Regel2'. In der Vereinbarung von Zeile 11 darf er nicht verwendet werden, aber in dem Prozeduraufruf auf Zeile 14 ist seine Benutzung legitim. Hinter der Vereinbarung von 'M' (Zeile 17) funktioniert dann auch das, was auf Zeile 11 noch verboten war (Konstantenvereinbarung auf Zeile 18). Ich wünsche mir zu Zeile 11 eine etwas genauere Fehlerausschrift als die Allerweltsmitteilung "Bezeichner nicht vereinbart". Aber die meisten Compiler sind nicht besser.

Die Menge der innerhalb einer Prozedur verwendbaren (d. h. gültigen) Bezeichner zerfällt in zwei disjunkte Teilmengen: die Menge der lokalen Bezeichner und den Rest. Diesen Rest bezeichnet man i. allg. als Menge der **globalen** Bezeichner. Ausgehend davon spricht man auch von lokalen und globalen Objekten. Objekte, die durch globale Bezeichner benannt werden, heissen globale Objekte.

Beispiel: In Prozedur 'p' von Programm 'Regel1' ist 'n' lokal; in 'q' ist dasselbe 'n' global.

Bisweilen werden nur jene globalen Objekte, die lokal zur Übersetzungseinheit (bisher Progammodul) sind, als global bezeichnet. Ich werde immer genau schreiben, was ich meine.

Gültigkeitsbereiche sind eine den Programmtext betreffende Eigenschaft: Wo im Quelltext kann ich welche Bezeichner verwenden? Bei der Ausführung von Programmen taucht die Frage nach der **Lebensdauer von Objekten**, insbesondere von formalen Parametern und Variablen, auf: Von wann bis wann existieren die Objekte im Rechner, d. h., wann wird ihnen Speicherplatz zugeteilt und wann wird er wieder zur anderweitigen Nutzung freigegeben? Es gibt zwei Fälle:

1. Variablen, die lokal zu einer Übersetzungseinheit (Programmodul, Baustein) sind, existieren vom Start des Programms bis zu dessen Ende. Ihre Lebensdauer erstreckt sich über die gesamte Abarbeitungszeit des Programms.
2. Die Lebensdauer der formalen Parameter und lokalen Variablen einer Prozedur beginnt mit dem Aufruf der Prozedur und reicht bis zum Ende der Prozedurabarbeitung.

Die Existenz der lokalen Objekte einer Prozedur ist nicht an die Prozedur, sondern an Aufrufe der Prozedur gebunden. Diese Feststellung ist wichtig für das Verständnis der Abläufe bei der Ausführung rekursiv definierter Prozeduren (s. Abschn. 7.4).

6. Standardtypen und Operationen

In der Programmiersprache MODULA-2 ist ein Datentyp (oder kürzer Typ) durch eine Menge von Werten und die Angabe der mit diesen Werten ausführbaren Operationen charakterisiert. Häufig nimmt man den zweiten Teil dieser Begriffsbestimmung - die möglichen Operationen - als implizit gegeben an und spricht einfach von einem Wertebereich. Die Sprachdefinition von MODULA-2 stellt dem Programmierer zehn "fertige" Datentypen zur Verfügung, die sogenannten Standardtypen, deren Bezeichner in Tafel 4.3 aufgelistet sind. Darüber hinaus bietet MODULA-2 Möglichkeiten zur Konstruktion weiterer Datentypen. Sie werden in den Abschnitten 9 und 12 ausfürlich dargestellt.
In den folgenden Typbeschreibungen bezeichne ich durch

MIN(typname) und MAX(typname)

das kleinste bzw. grösste Element des Standardtyps 'typname', der verschieden von BITSET sein muss. Diese Schreibweise ist auch in Programmen zulässig.

6.1. INTEGER, LONGINT

Der Standardtyp INTEGER besteht aus allen ganzen Zahlen x mit

MIN(INTEGER) <= x <= MAX(INTEGER)

(s. Tafel 6.1). Die ganzzahligen Literale ('ganze_zahl', S. 56), deren Wert kleiner oder gleich MAX(INTEGER) ist, sind die "elementaren" INTEGER-Konstanten. Im allgemeinen gilt die Gleichung

MIN(INTEGER) = -MAX(INTEGER)-1.

Mit Werten des Typs INTEGER sind die arithmetischen Grundrechenarten und Grössenvergleiche ausführbar. In Ausdrücken (s. Abschn. 7.1) werden sie durch Operationssymbole (Operatoren) dargestellt.

+	Addition	<	kleiner
-	Subtraktion	<=	kleiner oder gleich
*	Multiplikation	>	grösser
DIV	ganzzahlige Division	>=	grösser oder gleich
MOD	Divisionsrest	=	gleich
		#, <>	ungleich

a DIV b liefert den ganzzahligen Teil des Quotienten a durch b. Die Operation MOD ist nur für positive Operanden erklärt. a MOD b liefert den Rest der Division von a durch b (ggf. 0). Warnung: Obwohl in der Sprachdefinition die Gültigkeit der Beziehung

```
(a DIV b)*b + (a MOD b) = a
```

für positive a und b gefordert ist, sind nicht alle Realisierungen korrekt. Finden Sie deshalb heraus, wie die Ihnen zur Verfügung stehenden MODULA-2-Systeme sich bez. MOD und DIV verhalten!
Alle Vergleiche liefern ein Resultat vom Typ BOOLEAN. Er wird in Abschnitt 6.5 behandelt. Die Vergleichsoperatoren # und <> haben beide dieselbe Bedeutung. Die Operatoren + und - können auch einstellig verwendet werden. Sie bedeuten dann Identität bzw. Vorzeichenumkehr.
Neben den angeführten Operationssymbolen gibt es eine Reihe von Standardfunktionen, die auf Werte x des Typs INTEGER anwendbar sind.

```
ABS(x)    Absolutbetrag
ODD(x)    Test, ob x eine ungerade Zahl ist (Resultat BOOLEAN)
FLOAT(x)  Umwandlung des INTEGER-Werts x in einen REAL-Wert
```

Für Veränderungen des Werts von INTEGER-Variablen durch Addition bzw. Subtraktion sind zwei Standardprozeduren DEC und INC verfügbar. Beide können in jeweils zwei Formen verwendet werden. Seien v eine Variable vom Typ INTEGER und x ein INTEGER-Wert. Dann bewirken

```
DEC(v)    die Verringerung des Werts von v um 1
DEC(v,x)  die Verringerung des Werts von v um x
INC(v)    die Erhöhung des Werts von v um 1
INC(v,x)  die Erhöhung des Werts von v um x.
```

x kann auch negativ sein. In diesem Fall führt DEC zu einer Werterhöhung, und INC verringert den Wert von v.
Der Datentyp INTEGER ist eine endliche Wertemenge. Aus dieser Endlichkeit resultieren einige Probleme beim Rechnen mit INTEGER-Grössen, deren man sich bewusst sein muss. Aus dem Mathematikunterricht der Schule ist bekannt, dass im Bereich der ganzen Zahlen die Gleichung

```
(a + b) - c = a + (b - c)
```

für beliebige Werte a, b und c gilt. Wenn wir den Wertebereich auf INTEGER einschränken, dann muss diese Einschränkung nicht nur für das Endresultat einer Berechnungskette, sondern auch für jedes

Tafel 6.1. Übliche Grenzen der INTEGER- und CARDINAL-Wertebereiche auf 16- und 32-Bit-Rechnern

Grösse	Formel	wl = 16
MIN(LONGINT)	-2**(2*wl-1)	-2**31 = -2 147 483 648
MIN(INTEGER)	-2**(wl-1)	-2**15 = -32 768
MAX(INTEGER)	2**(wl-1) - 1	2**15-1 = 32 767
MAX(CARDINAL)	2**wl - 1	2**16-1 = 65 535
MAX(LONGINT)	2**(2*wl-1) - 1	2**31-1 = 2 147 483 647
MAX(LONGCARD)	2**(2*wl) - 1	2**32-1 = 4 294 967 295

Grösse	wl = 32
MIN(LONGINT)	-2**63 = -12 103 372 036 854 775 808
MIN(INTEGER)	-2**31 = -2 147 483 648
MAX(INTEGER)	2**31-1 = 2 147 483 647
MAX(CARDINAL)	2**32-1 = 4 294 967 296
MAX(LONGINT)	2**63-1 = 12 103 372 036 854 775 807
MAX(LONGCARD)	2**64-1 = 24 206 744 073 709 551 615

Teilergebnis erfüllt sein. Unter diesen Bedingungen kann es vorkommen, dass nur eine Seite unserer Gleichung ausrechenbar ist. Als Beispiel betrachten wir einen 16-Bit-Rechner mit

MAX(INTEGER) = 32767

sowie die Werte

a = 15000, b = 20000 und c = 25000.

Die Addition in der Klammer auf der linken Seite liefert 35000, und das ist kein INTEGER-Wert. Was geschieht in dieser Situation auf dem Rechner? Der günstigste Fall wäre ein Programmabbruch wegen "INTEGER-Überlauf" mit einem Hinweis auf die fragliche Programmstelle. Viele Systeme verhalten sich ganz anders. Der Überlauf bleibt unentdeckt; an Stelle der 35000 wird mit einer negativen Zahl (wahrscheinlich -30536 = 35000-65536) weitergerechnet, und das Endresultat ist sehr verwunderlich. Andere einschlägige Knackpunkte der INTEGER-Arithmetik sind die Anwendung der Funktion ABS auf die Zahl MIN(INTEGER) oder das "Überschreiten" der unteren bzw. oberen Grenze bei Verwendung der Standardprozeduren DEC und INC.

Ich habe es oft erlebt, dass erstaunliche Fehlverhaltensweisen meiner und studentischer Programme auf derartige Effekte zurückzuführen waren. Jeder Programmierer sollte schon beim Programmentwurf genau überlegen, welche Werte bei INTEGER-Berechnungen auftreten können. Manchmal genügt eine kleine Umstellung der Berechnungsreihenfolge zum Abwenden von Fehlern. Oft müssen die Wertebereiche der Eingabedaten rigoros eingeschränkt werden. Und in vielen Fällen benötigt man eben doch einen Rechner oder Compiler mit grösseren Zahlbereichen.

Für Objekte des Standardtyps LONGINT wird i. allg. doppelt soviel Speicherplatz bereitgestellt wie für INTEGER-Objekte. Das führt zu einem wesentlich vergrösserten Wertebereich (s. Tafel 6.1). Alle anderen Dinge sind fast genauso wie bei INTEGER; einzige Ausnahme: Die Standardfunktion FLOAT liefert für ein LONGINT-Argument ein LONGREAL-Resultat. MODULA-2 kennt, im Gegensatz zu vielen anderen Programmiersprachen, keine automatische Typanpassung:

> **Bei arithmetischen und bei Vergleichsoperationen müssen jeweils beide Operanden von demselben Standardtyp sein.**

Diese Forderung verbietet Verknüpfungen zwischen Objekten der Typen LONGINT und INTEGER. Beispielsweise ist die Multiplikation einer LONGINT-Grösse mit einer INTEGER-Grösse verboten.

Weil sich die Wertebereiche der beiden INTEGER-Typen überlappen, wird festgelegt, dass Konstanten, die im Durchschnitt von INTEGER und LONGINT liegen, zu beiden Typen gehören. Ich werde die Forderung nach Typidentität von Operanden und die Frage der gleichzeitig mehreren Typen angehörenden Konstanten bei der Besprechung der Typen CARDINAL und LONGCARD erneut aufgreifen müssen (s. Abschn. 6.2).

Es gibt viele MODULA-2-Systeme, die den Standardtyp LONGINT nicht unterstützen. Man merkt das daran, dass LONGINT kein vordefinierter Standardbezeichner ist. Versuche zur Verwendung dieses Typs werden von meinem MODULA-2-Compiler mit der Fehlermeldung

> **73: identifier not declared**

quittiert. Andere Compiler lassen zwar den Typbezeichner zu, aber der Wertebereich LONGINT ist identisch mit INTEGER. Es ist anzunehmen, dass bei allen neu auf den Softwaremarkt kommenden MODULA-2-Systemen echte LONG-Typen verfügbar sein werden.

6.2. CARDINAL, LONGCARD

Die Sprache MODULA-2 bietet dem Programmierer zwei weitere ganzzahlige Standardtypen an: CARDINAL und LONGCARD. Beide Wertebereiche enthalten nur nichtnegative Zahlen. Die Namenswahl geht auf die Mathematik zurück: Kardinalzahlen werden dort als Mächtigkeiten (Anzahl der Elemente) von Mengen eingeführt. Der Typ CARDINAL besteht aus den ganzen Zahlen x, die der Ungleichung

```
MIN(CARDINAL) = 0 <= x <= MAX(CARDINAL)
```

genügen. Die üblichen Maximalwerte sind Tafel 6.1 zu entnehmen. Der Wertebereich CARDINAL reicht "doppelt soweit" wie die nichtnegativen INTEGER-Werte; es gilt

```
MAX(CARDINAL) = 2*MAX(INTEGER) + 1.
```

Alle für INTEGER-Werte anwendbaren Operationen sind auch für CARDINAL-Werte zulässig. All das, was zum Verhältnis zwischen INTEGER und LONGINT gesagt wurde, gilt sinngemäss auch für CARDINAL und LONGCARD.

Die Wertebereiche der vier ganzzahligen Standardtypen von MODULA-2 überlappen sich. Die ganzzahligen Konstanten werden als zu jedem Typ, in den sie hineinpassen, gehörig betrachtet. Beispiel: Die nichtnegativen INTEGER-Konstanten sind auch Element der Wertebereiche CARDINAL, LONGINT und LONGCARD. Diese Verfahrensweise erlaubt dem Compiler eine gewisse Flexibilität beim Befolgen der auf S. 78 hervorgehobenen Forderung nach Typidentität der Operanden zweistelliger Verknüpfungen. Während z. B. die Subtraktion des Wertes einer Variablen vom Typ INTEGER von einer CARDINAL-Variablen (und umgekehrt) verboten ist, darf man die Konstante 17 sowohl von einer INTEGER- als auch von einer CARDINAL-Variablen abziehen. Im ersten Fall wird die 17 als INTEGER-Konstante und im zweiten als CARDINAL-Konstante aufgefasst.

Das Vorhandensein der CARDINAL-Typen hat technikgeschichtliche Ursachen. Niklaus Wirth schreibt in [Wirt87]: "Der Basistyp CARDINAL wurde in MODULA-2 eingeführt, um auf 16-Bit-Computern Adressrechnungen mit Werten zwischen 0 und 2**16 zu ermöglichen. Mit der Verbreitung von 32-Bit-Adressen in modernen Prozessoren ist die Notwendigkeit einer vorzeichenlosen Arithmetik praktisch verschwunden."

Wegen der lästigen Unverträglichkeit zwischen Operanden der Typen CARDINAL und INTEGER rate ich allen Programmierern, auf die Benutzung der CARDINAL-Typen zu verzichten. Leider kann diese Empfehlung aus Gründen, die in der Definition einiger Sprachelemente liegen, nicht 100%ig befolgt werden. Beispiele dafür sind die Standardfunktionen CHR und ORD (s. Typ CHAR, Abschn. 6.4) sowie HIGH (obere Grenze offener Felder, s. Abschn. 9.2.2). Jeder MODULA-2-Programmierer muss bez. dieser Dinge seinen eigenen Stil finden und diesen dann konsequent durchhalten. Ich bevorzuge z. B. in Zusammenhang mit HIGH eine Schreibweise, die mittels einer sogenannten Typtransferfunktion den Compiler explizit auffordert, einen CARDINAL-Wert als INTEGER-Wert aufzufassen. Ich werde das an entsprechender Stelle in den Abschnitten 9.2.2 und 13.2 erläutern.

6.3. REAL, LONGREAL

Die Wertebereiche der Standardtypen REAL und LONGREAL bestehen aus endlichen Teilmengen der rationalen Zahlen. Alle REAL-Werte x genügen der Ungleichung

```
MIN(REAL) <= x <= MAX(REAL).
```

Allerdings umfasst REAL nicht alle zwischen diesen Grenzen liegenden rationalen Zahlen; denn das wären unendlich viele. Die Syntaxregel 'reelle_zahl' (S. 56) beschreibt die elementaren Konstanten der beiden REAL-Typen. Die vielfältigen Probleme, die auf dem Computer beim Arbeiten mit rationalzahligen Approximationen reeller Zahlen auftreten, sind Gegenstand der "Numerischen Mathematik". Abschnitt 11 dieses Buches erläutert einige beim Rechnen mit REAL-Zahlen auftretende Erscheinungen, die der rechnerinternen Zahldarstellung geschuldet sind.

Für die Verknüpfung von REAL-Werten stehen folgende Operatoren zur Verfügung:

```
+   -   *   /   <   <=   >   >=   =   <>   #
```

Hierbei beschreibt "/" die Division. Alle anderen Operationssymbole haben die bereits bei INTEGER angegebene übliche Bedeutung. Zwei Standardfunktionen sind auf REAL-Werte anwendbar:

```
ABS(x)      Absolutbetrag des Wertes x    und
TRUNC(x)    Ganzzahliger Anteil von x für x >= 0.0
            (Resultattyp: CARDINAL).
```

Die Prozeduren DEC und INC können auf REAL-Variablen nicht angewendet werden. Zwischen den Funktionen FLOAT und TRUNC besteht für CARDINAL-Werte c folgender Zusammenhang:

```
TRUNC(FLOAT(c)) = c.
```

Wenn Werte von INTEGER- oder CARDINAL-Typen in Operationen mit REAL-Grössen einbezogen werden sollen, dann müssen sie unter Verwendung der Standardfunktion FLOAT explizit in einen REAL-Wert konvertiert werden. Literale notiert man natürlich am besten gleich in der richtigen Form. Seien r eine REAL- und i eine INTEGER-Variable. Mein Compiler reagiert sowohl auf

```
r + i   und   r + 5   als auch auf   r >= 123
```

mit der Anzeige des Quelltextfehlers

```
143: type incompatible operands.
```

An Stelle von 'i' ist 'FLOAT(i)' zu schreiben. Die beiden Literale wären durch '5.0' und '123.0' zu ersetzen.
Analog zu den INTEGER- und CARDINAL-Typen hat der Typ LONGREAL einen gegenüber REAL wesentlich vergrösserten Wertebereich. TRUNC konvertiert LONGREAL-Zahlen in LONGCARD-Werte.

6.4. CHAR

Schon lange werden Computer für weit über das Rechnen mit Zahlen hinausgehende Aufgaben eingesetzt. Textverarbeitungsprobleme fordern programmiersprachliche Möglichkeiten zum Manipulieren mit Zeichen. MODULA-2 trägt diesem Bedürfnis durch Bereitstellung des vordefinierten Typs CHAR Rechnung. Ein weiterer nichtnumerischer Standardtyp hat in den Programmiersprachen schon eine lange Tradition: Zur Beschreibung von Wahrheitswerten, die z. B. als Resultate von Vergleichen anfallen, gibt es den Typ BOOLEAN.
Der Standardtyp **CHAR** besteht aus den Zeichen des ASCII-Zeichensatzes (s. Tafel 4.4). Die Konstanten des Typs CHAR werden als Zeichenketten der Länge 1 oder als Oktalzahl mit nachgestelltem Grossbuchstaben "C" notiert (s. S. 58). Der Typ CHAR wird als entsprechend Tafel 4.4 geordnete Menge betrachtet. Demzufolge kann vom "Vorgänger" und vom "Nachfolger" eines CHAR-Wertes gesprochen werden, und die Vergleichsoperatoren

```
<   <=   >   >=   =   <>   #
```

sind ebenfalls erklärt. Die folgenden drei Standardfunktionen

stehen mit dem Typ CHAR in Verbindung.

CAP(x)	Umwandlung von Klein- in Grossbuchstaben Wenn x kein Kleinbuchstabe ist, dann wird x selbst als Resultat geliefert.
ORD(x)	Ordnungszahl des Zeichens x Das Resultat ist vom Typ CARDINAL.
CHR(c)	Zeichen mit der Ordnungszahl c c muss vom Typ CARDINAL sein.

Unter "Ordnungszahl" ist der jeweilige Zeichencode aus Tafel 4.4 zu verstehen. Offenbar gelten für CHAR-Werte x und solche CARDINAL-Zahlen, die kleiner als 128 sind, die Beziehungen

CHR(ORD(x)) = x und ORD(CHR(c)) = c.

Alle vier Formen der Standardprozeduren DEC und INC können auch auf Variablen v des Typs CHAR angewendet werden. Sei i ganzzahlig; dann bewirken

DEC(v)	den Übergang zum Vorgänger des Wertes von v und
DEC(v,i)	den Übergang zum i-ten Vorgänger des Wertes von v.

INC ist analog erklärt. Welche Reaktionen das MODULA-2-System auf das Verlassen des CHAR-Bereichs bei DEC, INC oder CHR zeigt, das steht hoffentlich in der Nutzerdokumentation.

Es gibt einen Trend, den Typ CHAR auf eine 8-Bit-Erweiterung des ASCII-Zeichensatzes auszudehnen und auf diese Weise Zeichen mit Ordnungszahlen zwischen 0 und 255 zuzulassen. Hier kann ich leider nur auf die Nutzerdokumentation Ihres MODULA-2-Systems bzw. auf geeignete Programmierexperimente verweisen.

6.5. BOOLEAN

Der Standardtyp BOOLEAN hat genau zwei Elemente, deren Namen die Standardbezeichner 'TRUE' und 'FALSE' sind. BOOLEAN ist eine geordnete Wertemenge: 'TRUE' kommt nach 'FALSE'. Die BOOLEAN-Werte werden als Wahrheitswerte (wahr/falsch) im Sinne der Aussagenlogik aufgefasst. Das Resultat arithmetischer Vergleichsoperationen hat den Typ BOOLEAN. Für BOOLEAN-Werte sind folgende zweistellige Operationen erklärt:

AND oder &	logisches "Und", Konjunktion
OR	logisches "Oder", Disjunktion
< <= > >= = <> #	Vergleiche.

Darüber hinaus gibt es die einstellige Operation

NOT oder ~ Verneinung, Negation.

Die Verknüpfung zweier BOOLEAN-Werte mit dem Operator AND liefert genau dann 'TRUE', wenn beide Operanden den Wert 'TRUE' haben, sonst 'FALSE'. Die OR-Verknüpfung ergibt nur dann das Resultat 'FALSE', wenn beide Operanden 'FALSE' sind. NOT kehrt den Wahrheitswert um: Aus 'TRUE' wird 'FALSE', und für 'FALSE' erhält man 'TRUE'. Die Operationssymbole AND und & sowie NOT und ~ sind jeweils gleichwertig.

Die Berechnung des Resultats von AND- und OR-Operationen unterscheidet sich bei MODULA-2 von der bei PASCAL üblichen Verfahrensweise. Ausgehend von der Beobachtung, dass das Endresultat bereits feststeht, wenn der erste Operand bei AND den Wert 'FALSE' bzw. bei OR den Wert 'TRUE' hat, wird in diesen Fällen der zweite Operand nicht mehr ermittelt. Seien A und B zwei Operanden vom Typ BOOLEAN. Die Berechnung von AND und OR ist exakt durch folgende "bedingte Ausdrücke", deren Bedeutung sofort klar sein sollte, gegeben:

A AND B wenn A dann B andernfalls FALSE

A OR B wenn A dann TRUE andernfalls B.

Hieraus ergibt sich, dass eine Konjunktion oder Disjunktion auch dann einen vernünftigen Wert haben kann, wenn ihr zweiter Operand gar nicht definiert ist. Man kann also Aussagen der Art

"eine Zahl x ist grösser als null"
AND
"das Resultat der Division einer Zahl y durch dieses x
ist kleiner als 20"

bequem formulieren: Für x = 0 wäre die Division "verboten"; aber sie wird nicht ausgeführt. Im Gegensatz dazu werden bei Vergleichen stets beide Operanden ausgewertet.

Die logischen Operationen "Implikation", "Äquivalenz" und "exklusives Oder" können sehr einfach durch Vergleichsoperatoren ausgedrückt werden.

Implikation	(aus A folgt B)	A <= B
Äquivalenz	(A genau dann, wenn B)	A = B
Exklusives Oder	(entweder A oder B)	A # B

Man überzeugt sich davon leicht durch Aufstellen der entsprechenden Wertetabellen. Beachten Sie, dass bei der Implikation das Vergleichssymbol gerade andersherum steht als der Implikationspfeil.

Worin unterscheidet sich diese "Realisierung" der Implikation von der ebenfalls möglichen Variante "NOT A OR B"?
Viele Anfänger benutzen in ihren ersten Programmen für den Test, ob eine BOOLEAN-Variable A den Wert 'TRUE' hat, den Operator '=' und schreiben

```
A = TRUE.
```

Das Resultat dieser Operation ist werteverlaufsgleich mit der Variablen A, d. h., man kann "= TRUE" getrost weglassen.
Der Typname BOOLEAN soll an den Juristen und Mathematiker G. Boole (1815-1864) erinnern, der Grundlegendes zur heutigen Form der Aussagenlogik beitrug. Boolesche Werte spielen in fast allen Programmen eine grosse Rolle, obwohl Variablen vom Typ BOOLEAN relativ selten sind. Sie steuern in "bedingten Anweisungen" (s. Abschn. 8.2 und 8.3) als Resultate von Vergleichen und deren Verknüpfung zu komplexeren Bedingungen die Auswahl der abzuarbeitenden Aktionen. Oft besteht kein Anlass, das Ergebnis derartiger Bedingungsüberprüfungen zusätzlich in einer BOOLEAN-Variablen aufzubewahren.

6.6. PROC

Der Standardtyp PROC beschreibt Prozeduren, deren abstrakte Signatur (s. Abschn. 5.1) leer ist. Mit anderen Worten: PROC beschreibt die parameterlosen reinen Prozeduren. Oder noch anders ausgedrückt: Die Konstanten des Typs PROC sind die parameterlosen reinen Prozeduren.
Ausgehend von der Einführung des Standardtyps PROC und den Möglichkeiten zur Vereinbarung weiterer Prozedurtypen (s. Abschn. 9.4) kennt MODULA-2 prozedurwertige Variablen und Parameter. Auch prozedurwertige Komponenten von ARRAY- und RECORD-Strukturen (s. Abschn. 9.2) sind ohne weiteres möglich.
Für Werte des Typs PROC sind keine Verknüpfungs- oder Vergleichsoperationen erklärt. Aber Wertzuweisungen an PROC-Variablen und Aktivierungen (Prozeduraufrufe) sind natürlich erlaubt. Durch

```
VAR
  E: PROC;
```

wird eine prozedurwertige Variable eingeführt, der z. B. mit der Ergibtanweisung

```
E := WriteLn;
```

die vom Baustein 'InOut' importierte Prozedur 'WriteLn' als Wert zugewiesen werden könnte. Nach dieser Wertzuweisung haben die Prozeduraufrufe

```
WriteLn;    und    E;
```

dieselbe Wirkung.
Das folgende kleine Beispiel 'Enden' (Programm 6.1) zeigt den gewinnbringenden Einsatz von Prozedurvariablen. Aufgabenstellung und Erläuterung befinden sich auf der nächsten Seite.

```
 1 MODULE Enden;
 2 (*-------------------------------------------------------------
 3   Demonstrationsbeispiel:    PROC-wertige Variablen
 4 -------------------------------------------------------------*)
 5 FROM InOut IMPORT Write, WriteString, WriteInt, WriteLn;
 6 (*-------------------------------------------------------------*)
 7 VAR  ZeilenNummer: INTEGER;   E: PROC (*Zeilenendeprozedur*);
 8 (*-------------------------------------------------------------*)
 9 PROCEDURE Nummern;
10 BEGIN   WriteLn;
11   INC(ZeilenNummer); WriteInt(ZeilenNummer,3); Write(' ');
12 END Nummern;
13
14 PROCEDURE Doppelt;
15 BEGIN
16   WriteLn; WriteLn;
17 END Doppelt;
18 (*-------------------------------------------------------------*)
19 PROCEDURE Arbeit;
20 BEGIN
21   E;    WriteString("aaa"); E;    WriteString("bbb"); E;
22 END Arbeit;
23 (*-------------------------------------------------------------*)
24 BEGIN   ZeilenNummer:=0;
25   E:=WriteLn; Arbeit;  E:=Doppelt; Arbeit;  E:=Nummern; Arbeit;
26 END Enden.
```

Programm 6.1. Benutzung prozedurwertiger Variablen

Die Aufgabe lautet:

> Ein Programm soll Texte zeilenweise verarbeiten und ausgeben. Für das Ausgabeverhalten am Zeilenende werden drei mögliche Varianten gewünscht: normaler Zeilenvorschub, doppelter Zeilenvorschub oder normaler Zeilenvorschub und Ausgabe einer Zeilennummer am Anfang der neuen Zeile.

Die Idee ist sehr einfach. Man vereinbare eine Variable vom Typ PROC und benutze sie im Verarbeitungsprogramm an all jenen Stellen, wo es um die Ausgabe eines Zeilenendes geht. Darüber hinaus werden drei parameterlose Prozeduren gebraucht, die jeweils eine der gewünschten Varianten realisieren. Am Verarbeitungsbeginn wird der Prozedurvariablen die entsprechende Zeilenendeprozedur zugewiesen. Programm 6.1 demonstriert die vorgeschlagene Lösung. "Echte" Textverarbeitung findet nicht statt. Die Prozedur 'Arbeit' enthält neben den hier interessierenden Aufrufen der Prozedurvariablen 'E' ein paar Zeichenkettenausgaben, die ein Verfolgen der gewünschten Effekte am Bildschirm ermöglichen.

Der zehnte Standardtyp von MODULA-2 - BITSET - beschreibt einen sogenannten Mengentyp. Er wird im Abschnitt 9.3 erläutert.

7. Ausdrücke und Konstantenausdrücke, Rekursion

Formelmässige Wertberechnungen werden in höheren Programmiersprachen durch sogenannte Ausdrücke beschrieben; ein Ausdruck ist eine Berechnungsvorschrift.
Im Mathematikunterricht der Schule gibt es für das Lösen von zusammengesetzten Rechenaufgaben den Merkspruch "Punktrechnung geht vor Strichrechnung". Er definiert zwei Operatorklassen (Punktrechnung: Multiplikation und Division; Strichrechnung: Addition und Subtraktion) und formuliert eine Vorrangregel. Die MODULA-2-Ausdrücke werden nach demselben Schema eingeführt.
Unter den elementaren Operanden, die in Ausdrücken auftreten, verdienen Funktionsaufrufe besondere Aufmerksamkeit. Die Regeln über Parameterlisten treffen auch für Aufrufe reiner Prozeduren zu.
Rekursive Funktions- und Prozeduraufrufe sind in MODULA-2 nicht verboten. Man muss sich nur daran gewöhnen. Ein Beispiel und eine ausführliche Diskussion sollen das unterstützen.

7.1. Ausdrücke

Analog zur Verfahrensweise mit Punkt- und Strichrechnung in der Schule werden zuerst vier Operatorklassen fixiert:

> Relationsoperatoren, Additionsoperatoren, Multiplikationsoperatoren und Negationsoperatoren.

Der Relationsoperator "IN" bedeutet "ist Element von". Er tritt im Zusammenhang mit Mengentypen auf und wird im Abschnitt 9 erläutert.

```
$    rel_op = "<" | "<=" | ">" | ">=" | "=" | "<>" | "#" | IN .
$    add_op = "+" | "-" | OR .
$    mul_op = "*" | "/" | DIV | MOD | AND | "&"
$    neg_op = "~" | NOT .
```

Die vierte Operatorklasse besteht nur aus den beiden Schreibweisen der Negation.
Negationsoperatoren treten ausschliesslich innerhalb von Elementarausdrücken (Faktoren) auf. Durch die Verknüpfung von Faktoren mit

Multiplikationsoperatoren entstehen Terme. Daraus kann man unter Verwendung von Additionsoperatoren einfache Ausdrücke bilden. Und auf der obersten Hierarchiestufe sind Ausdrücke als Vergleiche von zwei einfachen Ausdrücken beschrieben.

```
$    ausdruck = einfacher_ausdruck [rel_op einfacher_ausdruck]
$    einfacher_ausdruck = ["+"|"-"] term {add_op term}
$    term = faktor {mul_op faktor} .
$    faktor = elementaroperand | "(" ausdruck ")" | neg_op faktor .
$    elementaroperand = zahl | zeichenkette | menge
$                     | bezugnahme | funktionsaufruf .
+    bezugnahme = q_name .
```

Von den elementaren Operanden wurden die Literale ('zahl' und 'zeichenkette') bereits im Abschnitt 4.2 eingeführt. Syntax und Bedeutung von 'menge' folgen im Abschnitt 9.3. Als 'bezugnahme' lasse ich vorerst nur Konstanten- und Variablennamen zu, die ggf. durch einen Modulnamen qualifiziert sein können. In den Abschnitten 9.2 und 12.1 kommen noch Bezugnahmen auf Komponenten von ARRAY- und RECORD-Strukturen sowie auf dynamische Variablen hinzu. Funktionsaufrufe werden im nächsten Abschnitt behandelt.
Selbstverständlich ist nicht jede der Syntax 'ausdruck' entsprechende Symbolfolge ein gültiger MODULA-2-Ausdruck: Im Abschnitt 6 wird angegeben, welche Operatoren für den jeweiligen Datentyp anwendbar sind.
Für die Berechnungsreihenfolge - sprich: den Vorrang der vier Operatorklassen - ist festgelegt, dass mit Bezug auf die syntaktische Struktur

> zuerst Faktoren, dann Terme, danach einfache Ausdrücke und zuletzt Ausdrücke berechnet

werden. Bez. der Faktoren gilt, dass mit Klammerinhalten zu beginnen ist. Aufeinanderfolgende Additions- bzw. Multiplikationsoperatoren werden von links nach rechts abgearbeitet.
Für die Diskussion von Beispielen gehe ich von den Vereinbarungen

```
VAR
  i,j,k: INTEGER;   a: CARDINAL;   r,s: REAL;   z: CHAR;
  p,q: BOOLEAN;
```

aus. Folgende Ausdrücke sind syntaktisch und bez. der Operanden-

typen korrekt gebildet. Man beachte den inhaltlichen Unterschied zwischen den ersten beiden Ausdrücken!

```
i+j*k              (i+j)*k          r/s*0.5
z#'A'              a DIV 17         p AND (q OR (i>j))
```

Wenn die äusseren Klammern im letzten Beispiel weggelassen würden, dann wäre die Bedeutung sehr verändert.

Ich erinnere noch einmal an die spezielle Verfahrensweise beim Berechnen der Resultate von AND- und OR-Operationen (s. Abschn. 6.5): Wenn sich während der Berechnung herausstellt, dass 'p' den Wert 'FALSE' hat, dann wird

```
q  OR (i>j)
```

nicht ausgerechnet. Für 'p=TRUE' und 'q=TRUE' entfällt der Vergleich zwischen'i' und 'j'. Er wird nur ausgeführt, wenn 'p' den Wert 'TRUE' und 'q' den Wert 'FALSE' hat.

Typische Syntaxfehler sind Ungleichungsketten und vergessene Klammern beim Verknüpfen von Vergleichen.

```
i<=j<=k          a>17 OR r#s     z<='0' OR z>='9'
```

Mein Compiler äussert sich zu diesen Missbildungen salomonisch mit der Fehlernachricht

```
27: error in expression.
```

Das trifft zu, ist aber wenig hilfreich, wenn man nicht auf die Idee kommt, ein paar Klammern hinzuzufügen. Hier sind die korrekten Ausdrücke:

```
(i<=j)AND(j<=k)   (a>17)OR(r#s)   (z<='0')OR(z>='9').
```

Die Syntax von 'ausdruck' lässt beliebig kompliziert aufgebaute Gebilde zu:

> Ausdrücke werden durch Einschluss in runde Klammern zu Faktoren; einfache Ausdrücke und Terme können beliebig viele Additions- bzw. Multiplikationsoperatoren enthalten.

Wie verhalten sich die MODULA-2-Systeme dazu? Viele Compiler haben bez. des Aufbaus von Ausdrücken relativ enge Grenzen. Der ahnungslose Nutzer wird plötzlich mit einer Fehlerausschrift

```
implementation restriction: expression too complicated
```

konfrontiert. Sollte Ihnen dergleichen widerfahren, so führen Sie eine zusätzliche Variable ein. Etwa "die Hälfte" des kritisierten Ausdrucks sollte abgespalten und der neuen Hilfsvariablen als Wert zugewiesen werden (s. Wertzuweisung im Abschn. 8.1). Das folgende Beispiel veranschaulicht die vorgeschlagene Verfahrensweise. Alle

auftretenden Bezeichner sollen Namen von INTEGER-Variablen sein. Aus der Ergibtanweisung

```
a := a-10*(b+7*(c+d));
```

könnte man z. B.

```
h := 7*(c+d);  a := a-10*(b+h);
```

machen. Ich hoffe allerdings, dass kein MODULA-2-Compiler einen Ausdruck wie 'a-10*(b+7*(c+d))' als zu kompliziert zurückweist.

7.2. Konstantenausdrücke

Konstantenausdrücke sind im wesentlichen genauso aufgebaut wie Ausdrücke. Die Abweichungen liegen auf der Ebene der elementaren Operanden (s. Syntaxregel 'el_konst_operand'):

'q_name' ist auf benannte Konstanten eingeschränkt, und Funktionsaufrufe sind nicht möglich.

Im Abschnitt 9.3 wird der kleine Unterschied zwischen 'menge' und 'konst_menge' deutlich werden.

```
$    konstantenausdruck =
$      einfacher_konst_ausdruck [rel_op einfacher_konst_ausdruck]
$    einfacher_konst_ausdruck =
$      ["+"|"-"] konst_term {add_op konst_term}
$    konst_term = konst_faktor {mul_op konst_faktor}
$    konst_faktor = el_konst_operand
$                 | "(" konstantenausdruck ")"
$                 | neg_op konst_faktor .
$    el_konst_operand = zahl | zeichenkette | konst_menge
$                     | q_name .
   Der 'q_name' muss auf eine Konstante verweisen.
```

Konstantenausdrücke werden schon vom Compiler ausgewertet. Sie verursachen demzufolge keinen zusätzlichen Aufwand bei der Abarbeitung des Programms.

Die im Zusammenhang mit den Standardtypen (Abschn. 6) und den "gewöhnlichen" Ausdrücken (Abschn. 7.1) gemachten Bemerkungen zu erlaubten Operationen und zur Festlegung der Berechnungsreihenfolge sind auch hier gültig.

Im folgenden Beispiel für die gewinnbringende Verwendung "echter" Konstantenausdrücke hängt der Wert einer Konstanten von zwei anderen Konstantenwerten ab.

```
CONST
  hoehe = 12;
  breite = 7;
  anzahl = hoehe*breite;
```

Jede Änderung von 'hoehe' oder 'breite' ändert automatisch auch den Wert von 'anzahl'.

7.3. Funktions- und Prozeduraufrufe

Ein 'funktionsaufruf' ist ein Operand, dessen Typ der Resultattyp der gerufenen Funktionsprozedur ist. Wie wird der Wert solcher Operanden bestimmt?
Der Aufruf einer Funktion bewirkt die Vermittlung der Parameter und anschliessend die Abarbeitung der Anweisungsfolge im Funktionskörper. Diese Anweisungsfolge muss mindestens eine **RETURN-Anweisung** mit einem "Resultatausdruck" (Details s. Abschn. 8.1) enthalten.
Beispiel: Die Funktionsprozedur 'fibo' (Programm 7.1) hat zwei derartige Anweisungen:

```
RETURN 1                    (Zeile 10) und
RETURN fibo(n-2)+fibo(n-1)  (Zeile 11).
```

Die Abarbeitung der Funktion ist dann beendet, wenn die erste RETURN-Anweisung ausgeführt wurde:

> Der durch Berechnung des Resultatausdrucks ermittelte Wert wird als Resultat des Funktionsaufrufs "zurückgegeben",

d .h., er ist der Wert des durch den 'funktionsaufruf' beschriebenen Operanden.

```
$   funktionsaufruf = bezugnahme aktuelle_parameter .
```

Die 'bezugnahme' muss auf eine Funktionsprozedur bzw. auf eine funktionsprozedurwertige Variable oder ARRAY- oder RECORD-Komponente führen.

```
$   aktuelle_parameter = "(" [akt_par_liste] ")" .
$   akt_par_liste = akt_parameter {"," akt_parameter}
$   akt_parameter = ausdruck | bezugnahme .
```

Die im Aufruf anzugebenden aktuellen Parameter werden von links beginnend den in der Prozedurvereinbarung bzw. in der abstrakten Signatur des Prozedurtyps (s. Abschn. 9.4) festgelegten formalen Parametern zugeordnet. Deshalb muss die Anzahl übereinstimmen. Weitere Forderungen sind abhängig von der Parameterart:

1. Referenzparameter
   Als aktueller Parameter ist nur eine 'bezugnahme' auf ein Objekt erlaubt, das den gleichen Typ hat wie der formale Parameter.
2. Wertparameter
   Als aktueller Parameter ist ein Ausdruck zugelassen, dessen Wert "zuweisungskompatibel" mit dem Typ des formalen Parameters sein muss.

Zwei Typen heissen **zuweisungskompatibel**, wenn sie entweder beide gleich sind oder wenn der eine INTEGER und der andere CARDINAL (bzw. LONGINT und LONGCARD) ist. Eine umfassendere Definition der Zuweisungskompatibilität wird im Abschnitt 9.5 gegeben. Sie bezieht alle in MODULA-2 möglichen Datentypen ein.
Die Besonderheiten der Parametervermittlung bei ARRAY-OF-Konstruktionen im Typ des formalen Parameters sind im Abschnitt 9.2.2 nach der Einführung der ARRAY-Typen beschrieben.
Den Syntaxregeln 'funktionsaufruf' und 'aktuelle_parameter' ist zu entnehmen, dass in Aufrufen parameterloser Funktionsprozeduren stets ein leeres Klammernpaar geschrieben werden muss.
Da die Regeln für die Zuordnung von aktuellen zu formalen Parametern bei reinen Prozeduren dieselben sind wie bei Funktionsprozeduren, gebe ich die Syntax für 'prozeduraufruf' gleich mit an.

$ **prozeduraufruf = bezugnahme [aktuelle_parameter]**
Die 'bezugnahme' muss auf eine reine Prozedur bzw. auf eine entsprechende prozedurwertige Variable oder ARRAY- oder RECORD-Komponente führen.

Ein 'prozeduraufruf' ist eine 'anweisung' (s. Abschn. 8.1). Er bewirkt eine Parametervermittlung (falls die Prozedur Parameter hat) und die Abarbeitung der 'anweisungsfolge' im Prozedurkörper.
Aus den Syntaxregeln ergibt sich, dass eine reine parameterlose Prozedur sowohl ohne 'aktuelle_parameter' als auch mit einer leeren

Parameterliste (nur das Klammernpaar) gerufen werden kann. Beides ist gleichwertig. Ich ziehe es vor, keine leeren Klammern zu schreiben.

7.4. Rekursiv definierte Prozeduren

Die im Abschnitt 5.4 behandelten Regeln über die Gültigkeitsbereiche von Bezeichnern lassen es zu, dass der Name einer Prozedur innerhalb der Prozedur in einem Funktions- bzw. Prozeduraufruf verwendet wird. Wenn so etwas geschieht, dann spricht man von rekursiv definierten Prozeduren: Sie rufen sich selbst wieder auf. Damit aus diesem Selbstaufruf keine unendlich lange Kette entsteht, muss der Programmierer in rekursiven Prozeduren irgendwelche Bedingungen einbauen, die irgendwann einen weiteren rekursiven Aufruf ausschliessen.
Rekursive Prozeduren bieten sich immer dann als natürliches programmiersprachliches Ausdrucksmittel an, wenn Datenstrukturen oder Berechnungsvorschriften realisiert werden sollen, die ihrerseits eine rekursive Definition haben. Ein einfaches Beispiel dafür ist die Folge der Fibonaccischen Zahlen f[n] (n>0). Ihre Definition ist durch drei Gleichungen gegeben:

```
f[1] = 1
f[2] = 1
f[n] = f[n-2]+f[n-1] für n>2.
```

Eine Funktionsprozedur zur Berechnung der n-ten Fibonacci-Zahl müsste also für n<=2 den Wert 1 liefern. In allen anderen Fällen könnte sie um Berechnung der (n-2)-ten und der (n-1)-ten Fibonacci-Zahl "bitten" und das verlangte Resultat durch Addition beider Werte ermitteln. An "wen" soll man sich zur Berechnung der beiden kleineren Fibonaccischen Zahlen wenden? Es ist naheliegend, dafür gleich die hier beschriebene Funktionsprozedur zu verwenden, sie also rekursiv aufzurufen.
Die INTEGER-wertige Funktionsprozedur 'fibo' im Programm 7.1 verwirklicht diese Idee. Ich habe sie durch Fettdruck hervorgehoben.
Das Hauptprogramm erfragt den Index der gewünschten Zahl, lässt sie berechnen und schreibt sie aus. Die verwendeten Anweisungen wurden bereits beim einführenden Programmbeispiel 'Staedte' erläutert.

Darüber hinaus verweise ich auf Abschnitt 8. Die obere Grenze 23 im Aufruf von 'Fragen.ZahlFrage' habe ich gewählt, weil bei meinem MODULA-2-System die Ungleichungskette

```
f[23] < MAX(INTEGER) < f[24]
```

gilt. Versuchen Sie, den Ablauf beim Ruf 'fibo(4)' aufzuschreiben!

```
 1 MODULE FibonacciZahlen;
 2 (*-------------------------------------------------------------
 3   Eine rekursiv definierte Funktionsprozedur:
 4            Berechnung der Fibonaccischen Zahlenfolge
 5 ---------------------------------------------------------------*)
 6 IMPORT Fragen, InOut;
 7 (*-------------------------------------------------------------*)
 8 PROCEDURE fibo(n: INTEGER): INTEGER;
 9 BEGIN
10   IF n<=2 THEN RETURN 1 END;
11   RETURN fibo(n-2)+fibo(n-1) (*Hier steckt die Rekursivitaet*);
12 END fibo;
13 VAR index: INTEGER;
14 (*-------------------------------------------------------------*)
15 BEGIN
16   LOOP
17     Fragen.ZahlFrage("Index der Fibonacci-Zahl (0 fuer Stop)",
18                      0,23, index);
19     IF index=0 THEN EXIT END;
20     InOut.WriteInt(fibo(index),43); InOut.WriteLn;
21   END;
22 END FibonacciZahlen.
```

Programm 7.1. Rekursive Berechnung der Fibonacci-Zahlen

Beim Ausprobieren dieses Programms fällt auf, dass die Zeit zur Berechnung von f[n] für n>13 merklich grösser wird. Hier findet eine wahre Flut von Funktionsaufrufen statt. Und das kostet Zeit. Diese Beobachtung spricht nicht generell gegen die Verwendung rekursiv definierter Prozeduren. Sie spricht nur gegen mein Beispiel, wo dieselben Werte immer wieder neu berechnet werden.
Am Schluss von Abschnitt 8 werde ich ein "Rezept" angeben, wie

ausgehend von einer EBNF-Syntax Prozeduren zur "syntaxorientierten" Verarbeitung von Daten geschrieben werden können. Wenn in den EBNF-Regeln Rekursivitäten vorkommen, dann treten diese auch in den "erzeugten" Prozeduren auf. Weitere wichtige Einsatzfälle für rekursiv definierte Verfahren sind 'QuickSort' (s. Abschn. 10.4) und die Arbeit mit binären Bäumen (s. Abschn. 12.2).

Nachdem Sie sich davon überzeugt haben, dass die rekursiv definierte Funktionsprozedur 'fibo' das Verlangte leistet, soll nun erklärt werden, wie das abläuft. Zur schematischen Darstellung der Abarbeitung einer Prozedur 'xyz' mit dem Parameter 'p' notiere ich:

```
xyz(p)++++++++++R.
```

Vorn steht der Prozeduraufruf. Danach deuten die Kreuze das Verstreichen von Zeit an. Der Buchstabe "R" soll die Rückkehr aus der Prozedur symbolisieren. Wenn innerhalb einer Prozedur eine weitere Prozedur gerufen wird, dann füge ich in die Kreuzkette ein "A" (Aufruf) ein und schreibe die Abarbeitung der gerufenen Prozedur in die Zeile darunter. Über dem Rückkehr-R trage ich einen Stern ein. In der Zeit zwischen "A" und "*" ruht die Abarbeitung von 'xyz', aber sie ist noch nicht beendet. Ich notiere Striche an Stelle der Kreuze.

```
xyz(p)+A----------*+++++++++R
       abc(e)+++++++R
```

Auf diese Weise kann ich auch tiefere Aufrufverschachtelungen darstellen. Immer gilt: Von links nach rechts wächst die Zeit, und von oben nach unten wächst die Aufrufverschachtelungstiefe.

Bemerkung:

Bitte beachten Sie, dass die Verschachtelung der Prozedurvereinbarungen und die Verschachtelung von Prozeduraufrufen zwei sehr verschiedene Dinge sind.

Hier werden nur Aufrufe dargestellt. Wenn man die Berechnung der vierten Fibonacci-Zahl nach dem vorgeschlagenen Schema notiert, dann kommt etwa folgendes heraus:

```
fibo(4)+A-------*+++A-------------------------------*+++R
        fibo(2)+R   fibo(3)+A-------*+++A-------*+++R
                             fibo(1)+R   fibo(2)+R
```

Am Ende von Abschnitt 5.4 wurde mitgeteilt, dass die Lebensdauer

der lokalen Objekte einer Prozedur jeweils die Zeit der Prozedurabarbeitung (von Aufruf bis Rückkehr) umfasst. Man spricht in diesem Zusammenhang auch davon, dass für jeden Aufruf einer Prozedur jeweils eine "Instanz" der lokalen Grössen dieser Prozedur angelegt werde.
Die Prozedur 'fibo' hat ein lokales Objekt: den formalen Parameter 'n'. In dem Ablaufschema zum Aufruf 'fibo(4)' gibt es fünf Aufrufe der Prozedur 'fibo'. Daraus folgt: Es werden nacheinander fünf Instanzen des formalen Parameters 'n' "geboren". Jede Instanz von 'n' "lebt" dann bis zum Ende der entsprechenden Prozedurabarbeitung und "stirbt" bei der Rückkehr. Die Lebensdauern verschiedener Instanzen des lokalen Objekts 'n' überlappen sich.
So existieren z. B. während der gesamten zum Aufruf 'fibo(1)' gehörenden Abarbeitung von 'fibo' drei Instanzen des formalen Parameters 'n' parallel. Sie gehören zu den Aufrufen mit den Parametern 4, 3 und 1.
Abschliessende generelle Bermerkungen zu rekursiv definierten Prozeduren in Programmen:

1. Die Abarbeitung rekursiv definierter Prozeduren funktioniert, weil bei jedem Aufruf eine neue Instanz der lokalen Objekte der Prozedur angelegt wird, die bis zum Ende der zugehörigen Abarbeitung existiert.
2. Punkt 1 besagt u. a., dass bei jedem Aufruf einer Prozedur Speicherplatz für lokale Grössen zugeteilt wird. Wenn man versehentlich eine "unendliche" Rekursivität programmiert hat, dann wird das Programm nach einer gewissen Zeit wegen Speicherplatzmangels abgebrochen werden.
3. Es ist nützlich, sich die Abläufe bei rekursiv definierten Prozeduren an kleinen vollständig notierten Aufrufbeispielen klar zu machen. Es ist sinnlos, dies auch für komplizierte Abläufe zu versuchen. "Kompliziert" kann hier sowohl bedeuten, dass viele Aufrufe auftreten, als auch, dass die Rekursivität erst über mehrere dazwischenliegende Prozedurrufe entsteht.
4. Beim Entwurf rekursiver Prozeduren muss man sich wieder von dem gesamten Wissen über die Instanziierung der lokalen Objekte lösen und von der Vorstellung der vielen verschachtelten Aufrufe abstrahieren. Es kommt darauf an, Abbruchbedingungen als Schutz vor unendlichen Rekursionen einzubauen. Wenn man

das gemacht hat, dann sollte man beim Nachdenken über jeden konkreten rekursiven Aufruf von der Annahme "Die Prozedur, die ich hier rufe, leistet das Verlangte!" ausgehen.

Bez. Punkt 4 gibt es eine - nicht nur formale - Analogie zu mathematischen Beweisen nach der Methode der vollständigen Induktion. Man weiss, dass ein "Induktionsanfang" bewiesen werden muss. Aber beim Ausführen des Induktionsschlusses von n auf n+1 würde es gewöhnlich sehr stören, wenn man immer das "Zurückdrehen" der gesamten Schlusskette bis zum Induktionsanfang im Sinn hätte.

8. Anweisungen

Anweisungen beschreiben Aktionen. Einfache Anweisungen sind elementare unteilbare Aktionen. Strukturierte Anweisungen bieten unterschiedliche Mechanismen zur Steuerung der Abarbeitung von Anweisungsfolgen:

- Auswahl einer von mehreren Anweisungsfolgen und
- wiederholte Ausführung einer Anweisungsfolge (Zyklen).

Eine 'anweisungsfolge' (Syntaxregel auf S. 64) beschreibt die Nacheinanderausführung der durch Semikolon getrennten Anweisungen: Die Abarbeitung der jeweils nächsten Anweisung beginnt erst nach Abschluss der vorherigen. Überlappungen sind ausgeschlossen.

```
$    anweisung = [ einfache_anweisung | with_anweisung
$                | anweisungsauswahl | zyklusanweisung ]
```

Wegen der eckigen Klammern in der Syntaxregel ist als 'anweisung' auch die leere Symbolfolge zulässig. Man spricht in diesem Fall von der leeren Anweisung oder Leeranweisung, die natürlich nichts (also keine Aktion) bewirkt. Die Einführung von Leeranweisungen macht die Schreibweise von Programmen flexibler.
Beispielsweise kann am "Schluss" einer 'anweisungsfolge' immer ein Semikolon geschrieben werden. Syntaktisch steht dann dahinter noch ein Leeranweisung, die man nicht sieht und die nichts bewirkt, aber die Syntaxregel 'anweisungsfolge' passt wieder.
Die 'with_anweisung' steht in enger Beziehung zu RECORD-Typen und wird an der entsprechenden Stelle von Abschnitt 9.2.1 eingeführt.

8.1. Einfache Anweisungen

In der Sprache MODULA-2 gibt es vier elementare Anweisungen.

```
$    einfache_anweisung = wertzuweisung | prozeduraufruf
$                       | exit_anweisung | return_anweisung .
```

Variablen wurden im Abschnitt 5.2 als programmiersprachliche Objekte mit veränderlichem Wert eingeführt. Die elementare Operation zur

Wertveränderung ist die Wertzuweisung oder Ergibtanweisung. Der Wertzuweisungsoperator ":=" wird gelesen als "ergibt sich aus".

$ wertzuweisung = bezugnahme ":=" ausdruck .

Beispiele für Wertzuweisungen (Programm 2.2):

```
position := position+1;
maxNr := 0;
```

Der neue Wert des auf der linken Seite stehenden Objekts wird durch Berechnen des auf der rechten Seite notierten Ausdrucks ermittelt. Der Datentyp dieses Ausdrucks muss zuweisungskompatibel (s. S. 92) mit dem Typ der linken Seite sein.
Die Abarbeitung einer Ergibtanweisung besteht aus drei Schritten, die in der angegebenen Reihenfolge ausgeführt werden:

1. Ermitteln des durch die 'bezugnahme' bezeichneten Objekts
2. Berechnen des Ausdrucks
3. Zuweisen des Wertes an das unter 1 ermittelte Objekt.

Es ist zulässig, dass das links stehende Objekt auch in dem Ausdruck vorkommt. Beim Berechnen des Ausdrucks wird der alte Wert benutzt (vgl. auch die Diskussion auf den Seiten 24 und 25).
Entsprechend der Definition der Zuweisungskompatibilität dürfen INTEGER-Variablen CARDINAL-Werte zugewiesen werden und umgekehrt. Hier ist Vorsicht geboten! Die Wertzuweisung erfolgt ohne Konvertierung und ohne Prüfung auf Einhaltung von Wertebereichen; die interne Zahldarstellung wird unbesehen umgespeichert. Negative INTEGER-Werte werden auf diesem Wege zu CARDINAL-Zahlen, die zwischen MAX(INTEGER) und MAX(CARDINAL) liegen, und umgekehrt.

Prozeduraufrufe - manchmal werden sie auch Prozeduranweisungen genannt - sind bereits im Abschnitt 7.3 eingeführt und im Abschnitt 7.4 diskutiert worden.
Beispiele für Prozeduraufrufe:

```
ZahlFrage("   Einwohnerzahl", 1,30000, leute)
INC(i)
InOut.WriteInt(n*n,8)
```

Die EXIT-Anweisung besteht aus dem Schlüsselwort EXIT.

```
$    exit_anweisung = EXIT .
```

Sie darf nur als Komponente einer LOOP-Anweisung (s. Abschn. 8.3) auftreten und bewirkt dort das Verlassen der durch LOOP und END geklammerten Anweisungsfolge, d. h., die Verarbeitung wird hinter dem END fortgesetzt, es erfolgt ein "Sprung" unmittelbar hinter das Schleifenende.
Sinnloses (?) Prinzipbeispiel:

```
LOOP
  i:=2; EXIT; INC(i);
END;
i:=i*i;
```

Die Anweisung 'INC(i)' wird nicht erreicht. EXIT bewirkt den Sprung hinter das END zur Ergibtanweisung 'i:=i*i', und die INTEGER-Variable 'i' hat am Schluss den Wert 4.

Die RETURN-Anweisung von MODULA-2 dient zur Rückkehr aus einer Prozedur oder Übersetzungseinheit. Sie beendet die Abarbeitung der Anweisungsfolge des entsprechenden Prozedur- bzw. Modulblocks. Es gibt zwei Formen der RETURN-Anweisung.

```
$    return_anweisung = RETURN ausdruck
$                     | RETURN .
```

In Funktionsprozeduren wird die aus dem Schlüsselwort RETURN und einem nachgestellten Ausdruck bestehende Variante benutzt. Der Typ des Ausdrucks muss zuweisungskompatibel mit dem Resultattyp der Funktionsprozedur sein. Die Abarbeitung erfolgt in zwei Schritten:

1. Berechnen des Ausdrucks
2. Verlassen der Funktionsprozedur und Rückgabe des unter Punkt 1 berechneten Wertes als Funktionsresultat.

Jede Funktionsprozedur muss mindestens eine RETURN-Anweisung enthalten. Bei jeder Abarbeitung einer Funktionsprozedur muss eine RETURN-Anweisung "erreicht" werden. Wenn die Abarbeitung bis zum Prozedur-END fortschreitet, dann wird die Programmausführung als fehlerhaft abgebrochen.

Die Prozedur 'fibo' aus Programm 7.1 ist ein Beispiel für eine Funktionsprozedur mit mehreren RETURN-Anweisungen:

```
PROCEDURE fibo(n: INTEGER): INTEGER;
BEGIN
  IF n<=2 THEN RETURN 1 END;
  RETURN fibo(n-2)+fibo(n-1);
END fibo;
```

In reinen Prozeduren und innerhalb der Anweisungsfolge von Übersetzungseinheiten darf nur die kurze Form der RETURN-Anweisung verwendet werden. Sie besteht aus dem Schlüsselwort RETURN. Mehrere RETURN-Anweisungen sind möglich. Sobald eine bei der Abarbeitung "erreicht" wird, erfolgt die Rückkehr zur Aufrufstelle, bzw. die Ausführung des Programmoduls ist beendet.
Anders als bei Funktionsprozeduren fordert MODULA-2 bei reinen Prozeduren und in der Anweisungsfolge von Übersetzungseinheiten nicht unbedingt das Vorhandensein einer RETURN-Anweisung. Wenn die Abarbeitung ein Prozedur- oder Modul-END erreicht, dann wird so verfahren, als ob dort ein RETURN stehen würde.
MODULA-2 kennt keine Marken und keine GOTO-Anweisungen. Mancher FORTRAN- oder PASCAL-Programmierer vermisst diese Dinge sehr. Gemeinsam mit den "klassischen" Auswahl- und Zyklusanweisungen sollten EXIT- und RETURN-Anweisungen als strukturierter Ersatz dafür verstanden und benutzt werden. Nach kurzer Zeit beklagt man den erzwungenen Verzicht auf die GOTO-Anweisungen nicht mehr, und die Programme sind verständlicher geworden.

8.2. Auswahl einer Anweisungsfolge

Die Sprache MODULA-2 bietet zwei Mechanismen zum Auswählen und Abarbeiten "markierter" Anweisungsfolgen:

```
$    anweisungsauswahl = if_anweisung | case_anweisung
```

Bei der bedingten oder IF-Anweisung werden den zur Debatte stehenden Anweisungsfolgen Boolesche Ausdrücke vorangestellt. In einer Auswahl- oder CASE-Anweisung trägt jede Anweisungsfolge eine oder mehrere Konstanten als Kennzeichen.

Zur Ermittlung der abzuarbeitenden Anweisungsfolge werden bei IF-Anweisungen die BOOLEAN-wertigen Ausdrücke der Reihe nach ausgewertet, bis sich einmal 'TRUE' ergibt oder kein weiterer Ausdruck mehr da ist. CASE-Anweisungen berechnen den Wert eines Auswahlausdrucks und arbeiten die mit diesem Wert "markierte" Anweisungsfolge ab.
Die Syntax beider Anweisungen ist gegenüber PASCAL verändert. Bedingte Anweisungen können mehr als einen Testausdruck enthalten, und in Auswahl-Anweisungen darf ein ELSE-Zweig auftreten.

```
$    if_anweisung = IF ausdruck THEN anweisungsfolge
$                   { ELSIF ausdruck THEN anweisungsfolge }
$                   [ ELSE anweisungsfolge ]
$                   END .
```

Jeder 'ausdruck' muss vom Typ BOOLEAN sein.

Zu jedem IF gehört genau ein END. Wie ist die Ausführung einer bedingten Anweisung in MODULA-2 definiert?
Zunächst wird begonnen, die hinter IF und ELSIF stehenden Booleschen Ausdrücke der Reihe nach zu berechnen. Sobald einmal das Resultat 'TRUE' auftritt, wird die hinter dem zugehörigen THEN notierte 'anweisungsfolge' abgearbeitet, und die Ausführung der bedingten Anweisung ist beendet. Liefern alle Ausdrücke den Wert 'FALSE', dann wird entweder die hinter ELSE angegebene 'anweisungsfolge' ausgeführt, oder (kein ELSE-Teil vorhanden) die bedingte Anweisung ist ohne Ausführung einer 'anweisungsfolge' beendet.
Bedingte Anweisungen mit ELSE-Teil arbeiten also genau eine der aufgeschriebenen Anweisungsfolgen ab; wenn kein ELSE-Zweig angegeben ist, wird höchstens eine der Anweisungsfolgen ausgeführt.
Das folgende Beispiel einer IF-Anweisung lehnt sich an eine bei der Verarbeitung von Zeichenfolgen oft vorliegende Situation an. Abhängig von einem in einer CHAR-Varibalen (hier 'z') vorliegenden Zeichen sind bestimmte weitere Aktionen auszuführen (z. B. diverse Zählungen und Verarbeitungen).

```
IF (z>='0') AND (z<='9') THEN INC(n1); ZahlVerarbeiten;
ELSIF (z>='a') AND (z<='z') THEN INC(n2); WortVerarbeiten;
ELSIF (z='+') OR (z='-') THEN VorzeichenVerarbeiten;
ELSE MuellVerarbeiten;
END;
```

Beispiele für kürzere bedingte Anweisungen:

```
IF prozente>=100 THEN
  SchwarzeZahlenSchreiben;
ELSE
  RoteZahlenSchreiben;
END;
IF n<=2 THEN RETURN 1 END;
```

Allgemeine Bemerkungen:

1. Anders als bei PASCAL sind die Komponenten von bedingten Anweisungen in MODULA-2 Anweisungsfolgen und nicht einzelne Anweisungen. Aus diesem Grund ist das END am Schluss Pflicht.
2. Die von PASCAL bekannte Verbundanweisung (eine mit BEGIN und END geklammerte Anweisungsfolge) existiert in MODULA-2 nicht.
3. Die Aussage von Punkt 1 trifft sinngemäss auch auf CASE-, WHILE-, FOR- und WITH-Anweisungen zu. Beim "Umschreiben" von PASCAL-Programmen zu MODULA-2-Programmen muss man einerseits oft BEGIN streichen und andererseits END hinzufügen.
4. Überlegen Sie sich, wie Sie die bedingten Anweisungen in Ihren Programmtexten einrücken wollen! Und halten Sie sich dann auch immer daran! Tun Sie dasselbe für alle sprachlichen Ausdrucksmittel von MODULA-2!
5. Wenn RETURN- oder EXIT-Anweisungen in bedingten Anweisungen auftreten, dann notiere ich die nachfolgenden Anweisungen nicht als ELSE-Zweig, sondern beende die IF-Anweisung und schreibe den Rest gleichberechtigt dahinter. Betrachten Sie die nebeneinandergestellten Varianten der Anweisungsfolge in der Prozedur 'fibo'!

```
IF n<=2 THEN RETURN 1;
ELSE
  RETURN fibo(n-2)+fibo(n-1);
END;
```

```
IF n<=2 THEN RETURN 1 END;
RETURN fibo(n-2)+fibo(n-1);
```

Beide leisten dasselbe. Die rechte Form vermeidet eine Stufe der Verschachtelung von Anweisungen. Das bewährt sich besonders dann, wenn am Anfang einer Prozedur Rand- oder Fehlerfälle abgetestet und endgültig behandelt werden und danach die eigentliche Verarbeitung folgt.

Die im Abschnitt 11.2 abgedruckten Prozeduren zur Konvertierung von REAL-Zahlen sind entsprechend Bemerkung 5 programmiert.

Die Syntax der Auswahl- oder CASE-Anweisung ist umfangreicher als die von IF-Anweisungen.

```
$    case_anweisung = CASE ausdruck OF
$                       fall {"|" fall}
$                     [ ELSE anweisungsfolge ]
$                     END .
$    fall = [ liste_von_fallkennzeichen ":" anweisungsfolge ]
$    liste_von_fallkennzeichen =
$                     fallkennzeichen {"," fallkennzeichen}
$    fallkennzeichen =
$              konstantenausdruck [".." konstantenausdruck]
```

Der Typ des zwischen CASE und OF stehenden Auswahlausdrucks darf einer der Standardtypen INTEGER, LONGINT, CARDINAL, LONGCARD, CHAR und BOOLEAN oder ein Aufzählungstyp bzw. ein Teilbereichstyp (s. Abschnitt 9.1) sein. Alle auftretenden Konstantenausdrücke müssen typkompatibel zum Auswahlausdruck sein. Zwei Typen heissen kompatibel, wenn sie gleich sind oder wenn eine weitere Bedingung erfüllt ist, die im Abschnitt 9.5 erläutert wird. Ein 'fallkennzeichen' in Intervallform

konst_ausdruck ".." konst_ausdruck

beschreibt alle zwischen den Werten des linken und des rechten Konstantenausdrucks liegenden Werte; der rechte Wert darf nicht kleiner als der linke sein. Alle in den 'fallkennzeichen' einer CASE-Anweisung auftretenden Werte müssen voneinander verschieden sein. Insbesondere dürfen sich Intervalle nicht überlappen.
Die Ausführung einer CASE-Anweisung beginnt mit der Berechnung des Auswahlausdrucks. Wenn es eine Anweisungsfolge gibt, in deren 'liste_von_fallkennzeichen' der ermittelte Wert auftritt, so wird diese Anweisungsfolge abgearbeitet, und die CASE-Anweisung ist beendet. Tritt der Wert in keinem 'fallkennzeichen' auf, dann muss die hinter ELSE notierte 'anweisungsfolge' abgearbeitet werden.
Was geschieht, wenn kein 'fall' zutrifft und kein ELSE-Zweig angegeben wurde, das ist in der Sprachdefinition leider nicht festgelegt. Es sollte in der Nutzerdokumentation Ihres MODULA-2-Systems stehen. Man rechne mit einem Programmabbruch! Auf keinen Fall kann man sich darauf verlassen, dass in dieser Situation

analog zur IF-Anweisung gar nichts geschieht. Aber man kann ein solches Verhalten erzwingen, indem man einen "leeren" ELSE-Zweig hinschreibt.
Die auf S. 102 angegebene IF-Anweisung beschreibt einen Ablauf, der auch als CASE-Anweisung notiert werden kann.

```
CASE z OF
  '0'..'9': INC(n1); ZahlVerarbeiten;
| 'a'..'z': INC(n2); WortVerarbeiten;
| '+', '-': VorzeichenVerarbeiten;
ELSE MuellVerarbeiten;
END;
```

Wenn 'z' mit dem Zeichen "A" belegt ist, dann führt die Auswertung des Auswahlausdrucks sofort zum Prozeduraufruf 'MuellVerarbeiten'. Bei der IF-Anweisung ist dafür eine Kaskade von drei Booleschen Ausdrücken abzuarbeiten, und das dauert länger. Die Auswahlanweisung ist hier der bedingten Anweisung vorzuziehen. Nach meinem Geschmack drückt sie das Gewünschte auch klarer aus. Doch Vorsicht!
Sei 'n' eine INTEGER-Variable. Die Auswahlanweisung

```
CASE n OF
  0..9: WriteInt(n,1);
| -9..-1, 10..99: WriteInt(n,2);
| -99..-10, 100..999: WriteInt(n,3);
ELSE WriteInt(n,6);
END;
```

wird bei den meisten MODULA-2-Systemen nicht funktionieren. Mein Compiler meldet die Verletzung einer implementationsspezifischen Restriktion. Er hat eine obere Schranke für den Abstand zwischen dem kleinsten und dem grössten Wert in den 'fallkennzeichen' einer CASE-Anweisung. Häufig liegt diese Schranke bei 256.
Der Grund für diese Restriktion ist in der rechnerinternen Realisierung von Auswahlanweisungen zu suchen. Normalerweise wird eine Adresstabelle benutzt. Die Einträge in dieser Tabelle müssen ohne Lücken vom kleinsten bis zum grössten 'fallkennzeichen'-Wert reichen. Und sie soll nicht über Gebühr viel Speicherplatz beanspruchen. Deshalb wird die Anweisung

```
CASE n OF
  1..10: xyz; blahblah | 333: dreidreidrei ELSE abc;
END;
```

von vielen MODULA-2-Systemen zurückgewiesen, obwohl in den 'fall-kennzeichen' nur 11 Werte auftreten. Die "Spannweite" ist zu gross. Das passendere programmiersprachliche Ausdrucksmittel wäre hier eine bedingte Anweisung.
Die Syntax der Auswahlanweisung lässt leere Fälle zu. Das ergibt sich aus den eckigen Klammern auf der rechten Seite der EBNF-Regel 'fall'. Bei vielen älteren MODULA-2-Übersetzern ist das nicht erlaubt.
Leere Fälle bringen Vorteile für die einheitliche Notation von CASE-Anweisungen: Man kann sich auf eine feste Position für den senkrechten Strich festlegen (am Fallanfang oder am Fallende) und alle Fälle in gleicher Art aufschreiben.

```
CASE                          CASE
| 1: abc; def;                  1: abc; def |
| 2: ghi; jkl;                  2: ghi; jkl |
ELSE mno;                     ELSE mno;
END;                          END;
```

Beide Varianten enthalten je einen leeren Fall. Links befindet er sich unmittelbar hinter dem CASE; rechts steht er vor dem ELSE.
Die Semikolons am Schluss der ELSE-Zweige und der Fälle 1 und 2 im linken Beispiel sind nicht notwendig, aber praktisch. Wenn die Anweisungsfolgen durch Einfügen weiterer Zeilen ergänzt werden sollen, dann steht das als Trennung zwischen den Anweisungen erforderliche Semikolon schon da. Dass ich rechts am Ende der Fälle kein Semikolon geschrieben habe, ist eine reine Geschmacksfrage.

8.3. Zyklische Abarbeitung von Anweisungsfolgen

MODULA-2 bietet vier Anweisungen zur Konstruktion von Programmschleifen oder Zyklen an. Die allgemeinste Form ist die LOOP-Anweisung. Mit ihr können alle in MODULA-2 möglichen Schleifen ausgedrückt werden.

```
$    zyklusanweisung = loop_anweisung | while_anweisung
$                    | repeat_anweisung | for_anweisung .
```

Die Programmierpraxis hat gezeigt, dass es drei häufig vorkommende

Standardsituationen für den Aufbau von Zyklen gibt, die durch die Bereitstellung spezieller programmiersprachlicher Ausdrucksmittel unterstützt werden sollten: WHILE- und REPEAT- sowie FOR-Schleifen.

Die LOOP-Anweisung beschreibt die sich ständig wiederholende Ausführung einer Anweisungsfolge. Wenn die letzte Anweisung ausgeführt ist, dann wird sofort wieder mit der ersten begonnen.

```
$    loop_anweisung = LOOP anweisungsfolge END .
```

Für den Ausstieg aus dieser Schleife stehen dem Programmierer zwei Möglichkeiten zur Verfügung:

- Abarbeiten einer EXIT-Anweisung oder
- Ausführen einer RETURN-Anweisung

innerhalb der 'anweisungsfolge'. Im Normalfall werden solche Anweisungen als Bestandteil von bedingten oder Auswahlanweisungen auftreten.

Die als **Programm 8.1** auf der folgenden Seite abgedruckte Prozedur 'LiesInteger' enthält zwei LOOP-Anweisungen. Sie verarbeitet eine Folge von Zeichen, die durch Rufe der 'InOut'-Prozedur 'Read' bereitgestellt werden.

Nach der Zuweisung eines Anfangswerts für den Referenzparameter 'wert' (Zeile 5) überliest die auf Zeile 6 notierte LOOP-Anweisung Steuer- und Leerzeichen: Die EXIT-Anweisung wird erst dann erreicht, wenn 'InOut.Read' ein Zeichen geliefert hat, das grösser als 40C (Leerzeichen, s. Tafel 4.4) ist.

Nun wird eine ggf. vorzeichenbehaftete ganze Zahl erwartet. Die bedingte Anweisung auf den Zeilen 7 bis 9 stellt den Schalter "negatives Vorzeichen angegeben" (d. h. die Variable 'negativ'). Falls ein Vorzeichen auftrat, muss das nächste Zeichen eingelesen werden; aber vorher ist das Vorzeichen auszuschreiben.

Die Schleife auf den Zeilen 10 bis 13 akzeptiert eine Ziffernfolge und interpretiert sie als Dezimaldarstellung einer ganzen Zahl. Man sagt: Die Ziffernfolge wird in den durch sie repräsentierten INTEGER-Wert konvertiert. Wenn keine Ziffer (mehr) vorliegt, dann geht es per EXIT nach Zeile 14. Ansonsten wird der bisher aufgelaufene 'wert' mit 10 multipliziert, und man addiert den durch die aktuelle Ziffer dargestellten INTEGER-Wert (Zeile 11). Mit dem Ausschreiben

```
 1 PROCEDURE LiesInteger(VAR wert: INTEGER);
 2 VAR
 3   z: CHAR;  negativ: BOOLEAN;
 4 BEGIN
 5   wert := 0;
 6   LOOP InOut.Read(z); IF z>" " THEN EXIT END END;
 7   IF (z="+")OR(z="-") THEN
 8     negativ := z="-"; InOut.Write(z); InOut.Read(z);
 9   ELSE negativ := FALSE END;
10   LOOP IF (z<"0") OR (z>"9") THEN EXIT END;
11     wert := wert*10 + INTEGER(ORD(z)-ORD("0"));
12     InOut.Write(z); InOut.Read(z);
13   END;
14   IF negativ THEN wert := -wert END;
15 END LiesInteger;
```

Programm 8.1. LOOP-Anweisungen beim Einlesen von Zahlen

der gerade verarbeiteten Ziffer und dem Einlesen eines neuen Zeichens (Zeile 12) werden die Voraussetzungen für die nächste "Runde" in der durch LOOP und END begrenzten Schleife geschaffen.
Der durch die Ziffer in 'z' repräsentierte INTEGER-Wert wurde mittels des "Funktionsaufrufs"

```
INTEGER(ORD(z)-ORD("0"))
```

notiert. Aus der Definition der ORD-Funktion für CHAR-Werte (s. Abschn. 6.4) und dem Umstand, dass die Ziffern von '0' bis '9' im Typ CHAR direkt aufeinanderfolgen, ergibt sich: ORD(z)-ORD("0") ist der CARDINAL-Wert der in 'z' gespeicherten Ziffer. Wegen der Forderung nach Typgleichheit der Operanden kann ich dieses Resultat nicht ohne weiteres zu 'wert*10' addieren. MODULA-2 bietet die Möglichkeit, Typbezeichner als Namen von Typtransferfunktionen zu verwenden. Durch den "Funktionsaufruf" INTEGER(x) wird der Compiler ausgetrickst. Er arbeitet mit dem Wert des Ausdrucks x so weiter, als hätte dieser den Typ INTEGER. Eine Konvertierung findet nicht statt (vgl. die Bemerkungen zu Wertzuweisungen zwischen INTEGER und CARDINAL auf S. 99). Verwenden Sie solche Ausdrucksmittel bitte nur dann, wenn es anders wirklich nicht geht!

Die LOOP-Anweisungen im Programm 8.1 zeigen zwei Standardfälle der Schleifenbildung: In einem Fall (Zeile 10) liegt der Test der Abbruchbedingung am Anfang der wiederholt abzuarbeitenden Anweisungsfolge; im anderen Fall liegt er am Ende (Zeile 6). Für diese Standardsituationen wurden "abkürzende" Schreibweisen definiert.

```
$    while_anweisung = WHILE ausdruck DO anweisungsfolge END .
   Der 'ausdruck' muss vom Typ BOOLEAN sein.
$    repeat_anweisung = REPEAT anweisungsfolge UNTIL ausdruck .
   Der 'ausdruck' muss vom Typ BOOLEAN sein.
```

Die LOOP-Anweisung auf den Zeilen 10 bis 13 von Programm 8.1 kann günstig als WHILE-Anweisung ausgedrückt werden.

```
WHILE (z>="0")AND(z<="9") DO
  wert := wert*10 + INTEGER(ORD(z)-ORD("0"));
  InOut.Write(z); InOut.Read(z);
END;
```

Wenn der hinter WHILE stehende Boolesche Ausdruck den Wert 'TRUE' liefert, dann wird die 'anweisungsfolge' abgearbeitet, und die Schleife beginnt danach erneut mit der Auswertung des Ausdrucks. Im anderen Fall (Resultat 'FALSE') ist die Ausführung der WHILE-Anweisung beendet. Es kann also passieren, dass die 'anweisungsfolge' überhaupt nicht abgearbeitet wird.
Die Bedingung in der als Beispiel angegebenen WHILE-Anweisung ist die Negation des in Zeile 10 von Programm 8.1 stehenden Ausdrucks. Vor EXIT wurde gefragt, ob abgebrochen werden soll. In der WHILE-Anweisung wird gefragt, ob fortzusetzen ist.

In einer REPEAT-Anweisung wird jeweils nach Ausführung der 'anweisungsfolge' getestet, ob aufgehört werden soll. Wenn die Auswertung der Abbruchbedingung ('ausdruck') den Wert 'FALSE' liefert, dann ist die Anweisungsfolge erneut abzuarbeiten. Andernfalls (Resultat 'TRUE') wird die Ausführung der REPEAT-Anweisung beendet.
Die Schleife in Zeile 6 der Prozedur 'LiesInteger' sollte man besser als REPEAT-Anweisung formulieren.

```
REPEAT InOut.Read(z) UNTIL z>" ";
```

Allgemeine LOOP-Anweisungen sind immer dann zu verwenden, wenn mehrere Abbruchbedingungen an unterschiedlichen Stellen der den

Körper der Schleife bildenden Anweisungsfolge geprüft werden müssen oder wenn der Abbruchtest weder am Anfang noch am Ende der Schleife liegt. WHILE- und REPEAT-Anweisungen können nicht mit einer EXIT-Anweisung beendet werden. (Es sei denn, sie stehen innerhalb einer LOOP-Anweisung.)

Sehr häufig müssen Schleifen programmiert werden, bei denen schon vorher ausgerechnet werden kann oder bekannt ist, wie viele Durchläufe erforderlich sind. Manchmal wird der "Rundenzähler" innerhalb der zu wiederholenden Anweisungsfolge benötigt. Oft ist es praktisch, nicht in Einerschritten zu zählen, sondern mit einer anderen Schrittweite, vielleicht auch rückwärts. Für solche Situationen gibt es in MODULA-2 die Laufanweisung (FOR-Anweisung).

```
$    for_anweisung = FOR laufvariable ":=" anfangswert TO endwert
$                    [BY schrittweite] DO anweisungsfolge END .
$    laufvariable = bezeichner .
   Der 'bezeichner' muss der Name einer lokalen Variablen des
   Blocks sein, in dessen Anweisungsteil die 'for_anweisung' steht.
$    anfangswert = ausdruck .
$    endwert = ausdruck .
$    schrittweite = konstantenausdruck .
```

Der Typ der 'laufvariablen' darf einer der Standardtypen INTEGER, LONGINT, CARDINAL, LONGCARD, CHAR und BOOLEAN oder ein Aufzählungstyp bzw. ein Teilbereichstyp (s. Abschn. 9.1) sein. Der 'anfangswert' und der 'endwert' müssen typkompatibel zur Laufvariablen sein. Als 'schrittweite' sind INTEGER-, LONGINT-, CARDINAL- und LONGCARD-Werte erlaubt. Wenn keine Schrittweite angegeben ist, dann wird der Wert 1 angenommen.
Die Ausführung einer FOR-Anweisung beginnt mit den folgenden drei Schritten:

1. Berechnen des Anfangswertausdrucks
2. Zuweisen des unter 1 ermittelten Wertes an die Laufvariable
3. Berechnen des Endwertausdrucks und "Merken" des Resultats

Nach diesen Vorbereitungen folgt der eigentliche Zyklus. Wenn der aktuelle Wert der Laufvariablen bei positiver Schrittweite nicht grösser bzw. bei negativer Schrittweite nicht kleiner als der

gemerkte Endwert ist, dann wird

1. die 'anweisungsfolge' ausgeführt
2. der Wert der Laufvariablen um die Schrittweite verändert
3. beim Test "Laufvariable gegen Endwert" fortgesetzt.

Andernfalls (Wert der Laufvariablen zu gross bzw. zu klein) ist die Abarbeitung der Laufanweisung beendet. Es wird hinter dem END fortgesetzt.

Ähnlich wie bei der WHILE-Anweisung kann es passieren, dass die 'anweisungsfolge' überhaupt nicht abgearbeitet wird. Im Gegensatz zu Anfangs- und Endwert muss die Schrittweite nicht für jede Abarbeitung einer Laufanweisung neu berechnet werden; das erledigt einmalig der Compiler.

Sei 'n' eine INTEGER-Grösse. Eine Prozedur zur Ausgabe der Quadrate der ersten 'n' durch drei teilbaren Zahlen in absteigender Reihenfolge könnte sicher mittels einer LOOP-Anweisung geschrieben werden. Sehr gut geht es mit einer Laufanweisung, wie die folgende Prozedur 'q3' zeigt.

```
PROCEDURE q3(n: INTEGER);
VAR i: INTEGER;
BEGIN
  FOR i:=n*3 TO 3 BY -3 DO
    InOut.WriteInt(i*i,10); InOut.WriteLn;
  END;
END q3;
```

Bemerkungen zur Laufvariablen in und nach Laufanweisungen:

1. Die Sprachdefinition legt nicht fest, welchen Wert die Laufvariable am Ende der Ausführung einer Laufanweisung hat. Man muss deshalb davon ausgehen, dass ihr Wert undefiniert ist, d. h., man sollte ihn nicht benutzen wollen.
2. Innerhalb der Laufanweisung sollte man nur Lesezugriffe auf die Laufvariable ausführen. Wenn der Wert der Laufvariablen verändert wird, dann entstehen unübersichtliche Abläufe. Einige Implementationen von MODULA-2 verbieten explizite Wertzuweisungen an die Laufvariable.

Im folgenden Beispiel (**Programm 8.2**) tritt eine Laufvariable vom Typ CHAR auf: Das Programm 'ZeichenCodes' druckt alle Nicht-Steuerzeichen des ASCII-Zeichensatzes und den zugehörigen Code in aufsteigender Reihenfolge.

```
 1 MODULE ZeichenCodes;
 2 (*-------------------------------------------------------------
 3   Beispiel mit einer Laufvariablen vom Typ CHAR:
 4           Ausschreiben aller Nicht-Steuerzeichen des
 5           ASCII-Zeichensatzes
 6 -------------------------------------------------------------*)
 7 FROM InOut IMPORT WriteLn, WriteInt, Write;
 8 (*-------------------------------------------------------------*)
 9 VAR
10   zeichen: CHAR;
11 BEGIN
12   FOR zeichen:=40C TO CHR(126) DO
13     IF ORD(zeichen) MOD 8 = 0 THEN WriteLn END;
14     WriteInt(ORD(zeichen),6); Write(" "); Write(zeichen);
15   END;
16   WriteLn;
17 END ZeichenCodes.
```

Programm 8.2. Verwendung einer CHAR-Laufvariablen

Bemerkungen zum Programm 8.2 'ZeichenCodes':

1. Das Programm "weiss" nur, dass 40C das erste Nicht-Steuerzeichen ist, dass das letzte ASCII-Zeichen die Ordnungszahl 127 hat und dass dieses letzte Zeichen ein Steuerzeichen ist. Dementsprechend sind Anfangs- und Endwert der Laufanweisung notiert.
2. Die bedingte Anweisung in Zeile 13 schreibt vor jedem Zeichen, dessen Ordnungszahl durch 8 teilbar, ist ein Zeilenende. Die Ausgabe beginnt deshalb mit einer Leerzeile. Danach stehen jeweils acht Code/Zeichen-Paare auf einer Zeile.
3. Im ersten Parameter von 'WriteInt' (Zeile 14) treten die bei Programm 8.1 diskutierten INTEGER/CARDINAL-Schwierigkeiten nicht auf, weil bei Wertparametern Zuweisungskompatibilität gefordert ist.

Wenn auf Ihrem Rechner und in Ihrem MODULA-2-System ein anderer, evtl. auf 256 Zeichen erweiterter Typ CHAR verfügbar sein sollte, dann brauchen Sie nur die Angaben für Anfangs- und Endwert (Zeile 12) Ihren Bedingungen anzupassen.

8.4. Ein "Rezept" für syntaxorientierte Programme

Im Abschnitt 4 wurde die EBNF als Hilfsmittel zum Formulieren von Regeln für das Aneinanderreihen einzelner Symbole zu Symbolfolgen eingeführt. Ich möchte in diesem Abschnitt eine softwaretechnologisch wichtige Anwendung formaler Grammatiken am Beispiel der EBNF darstellen:

> Die Entwicklung syntaxorientierter Programme. Bisweilen spricht man auch von "syntaxorientiertem Programmentwurf".

Oft lässt sich die Gestalt von Eingabe- und/oder Ausgabedaten eines Programms sehr klar in EBNF ausdrücken. Wenn eine solche Datenbeschreibung vorliegt, dann ist es möglich, aus den Regeln in systematischer Weise Prozeduren zur Prüfung der syntaktischen Korrektheit bzw. zur regelgerechten Datenausgabe zu konstruieren. Diese abgeleiteten Prozeduren können als "Gerippe" der eigentlich beabsichtigten Programme dienen: Man füge die mit den einzelnen Datenelementen auszuführenden Aktionen (oft semantische Aktionen genannt) nachträglich an den passenden Stellen als "Fleisch" ein.
Am Anfang einer systematischen Entwicklung der als Programm 8.1 abgedruckten Prozedur 'LiesInteger' steht bei dieser Entwurfsmethode die Prozedur:

```
PROCEDURE LiesInteger;
VAR  z: CHAR;
BEGIN
  REPEAT InOut.Read(z) UNTIL z>" ";
  IF (z="+")OR(z="-") THEN
    InOut.Read(z);
  END;
  WHILE (z>="0") AND (z<="9") DO
    InOut.Read(z);
  END;
END LiesInteger;
```

Die später eingefügten semantischen Aktionen betreffen das Ausschreiben der Ziffern und des Vorzeichens, die Verarbeitung des Vorzeichens und das Berechnen des Wertes der Zahl.
Ich werde mich in diesem Abschnitt auf solche Programme beschränken, die Eingabedaten syntaktisch prüfen und weiterverarbeiten. Die

syntaxorientierte Konstruktion von Ausgabeprogrammen ist in ähnlicher Weise möglich. Programme, die Eingabedaten syntaktisch prüfen und korrekte Eingabeströme akzeptieren, während fehlerhafte Dinge zurückgewiesen werden, heissen Parser (engl.). Es geht also um die systematische Konstruktion von Parsern aus Syntaxdarstellungen in EBNF.

Die Behandlung dieser Thematik als Abschluss eines Kapitels über Anweisungen bietet sich deshalb an, weil den einzelnen Ausdrucksmitteln der EBNF (Alternativen, eckige Klammern usw.) sehr direkt bestimmte Anweisungen (CASE-Anweisung, IF-Anweisung usw.) entsprechen. Vertiefende Literaturstellen zum syntaxorientierten Programmentwurf sind u. a. [Wirt77], [Pomb84] und [LiTr87].

Die systematische Konstruktion eines Parsers zu einer in EBNF vorgegebenen Syntaxbeschreibung liefert für jedes Hilfssymbol eine Prozedur, deren Inhalt durch die rechte Seite der zugehörigen Regel bestimmt ist. Eine globale Variable und zwei Prozeduren stellen die Verbindung der Parserprozeduren mit der "Aussenwelt" her:

1. die Variable 'sy', das jeweils aktuelle Terminalsymbol
2. die Prozedur 'Lesen', die das jeweils nächste Terminalsymbol liefert und in der Variablen 'sy' ablegt
3. eine Prozedur 'Fehler', die aufgerufen wird, wenn das aktuelle Terminalsymbol nicht "passt"

Als Terminalsymbole lasse ich vorerst nur einzelne Zeichen zu. Der generelle Aufbau eines Parsers bekommt damit folgende Gestalt:

```
MODULE Parser;
(*-----------------------------------*)
IMPORT InOut;  VAR  sy: CHAR;
(*-----------------------------------*)
PROCEDURE Lesen;
  BEGIN InOut.Read(sy) END Lesen;
PROCEDURE Fehler;
  BEGIN HALT END Fehler;
(*-----------------------------------
  Vereinbarung der Parser-Prozeduren
-----------------------------------*)
BEGIN
  Lesen; s;
END Parser.
```

Die Fehlerbehandlungsprozedur arbeitet vorerst mit der "Brechstange": Abbruch der Arbeit durch Ruf der Standardprozedur 'HALT'. Im Anweisungsteil stellt ein Aufruf von 'Lesen' das erste Terminalsymbol bereit. Anschliessend wird die zum Satzsymbol s der Grammatik gehörende Parserprozedur 's' aufgerufen. Und das ist es dann schon gewesen!

Gegeben seien eine Menge von Hilfssymbolen und eine Menge von Terminalsymbolen. Beide Mengen seien durchschnittsfremd (d. h., sie dürfen keine gemeinsamen Elemente besitzen).

Als Grundlage für die Beschreibung der Transformation von EBNF-Regeln in Parserprozeduren ist eine induktive Definition der Syntaxausdrücke nützlich:

1. Jedes Terminalsymbol ist ein Syntaxausdruck. Jedes Hilfssymbol ist ein Syntaxausdruck.
2. Wenn a1, a2, ..., an und a Syntaxausdrücke sind, dann sind auch a1 a2 ... an, a1 | a2 | | an, [a], { a } und (a) Syntaxausdrücke.
3. Eine Symbolfolge ist nur dann eine Syntaxausdruck, wenn das auf Grund von 1 und 2 der Fall ist.

"Symbol" bedeutet in Punkt 3 Hilfssymbol, Terminalsymbol, Klammer oder senkrechter Strich.

Nun sei eine konkrete Syntaxbeschreibung in EBNF gegeben. Für jedes Hilfssymbol existiere genau eine Regel. (Diese Forderung ist keine Einschränkung, da mehrere Regeln mit gleicher linker Seite unter Verwendung des Alternativoperators "|" zu einer einzigen zusammengefasst werden können.) Für alle Syntaxausdrücke a, die in der vorgegebenen Syntax als Komponenten von Alternativen auftreten, benötigen wir die Menge FIRST(a).

> Mit FIRST(a) wird die Menge aller jener Terminalsymbole bezeichnet, die am Anfang einer aus a ableitbaren Symbolfolge stehen können.

Die Konstruktion des Parsers beginnt mit der Einführung von Prozeduren: Für jedes Hilfssymbol wird eine parameterlose reine Prozedur vereinbart, deren Name gleich dem Hilfssymbol ist. Wenn das Hilfssymbol Unterstreichungszeichen ("_") enthält, dann werden sie weggelassen.

Der innere Aufbau der einzelnen Prozeduren ergibt sich aus den rechten Seiten der jeweiligen Regeln. Jeder Syntaxausdruck a wird

in ein Programmstück P[a] übersetzt. Ich notiere die einzelnen Übersetzungsregeln in der Form

```
a   --->   P[a].
```

Innerhalb der Programmstücke P[a] treten manchmal Tests auf, ob der Wert von 'sy' in FIRST(a) enthalten ist. Sei FIRST(a) die Menge {t1, t2, ... tm} mit m>0. Ich schreibe in den Übersetzungsvorschriften

```
or[a]   für   (sy=t1) OR (sy=t2) OR ... OR (sy=tm)
```

und

```
fk[a]   für   t1, t2, ..., tm.
```

or[a] tritt als Bedingung bei IF- oder WHILE-Anweisungen auf. fk[a] ist ein Fallkennzeichen in CASE-Anweisungen. Die grundlegenden Transformationsvorschriften folgen der induktiven Definition von Syntaxausdrücken:

1. Jedes Terminalsymbol t wird in eine bedingte Anweisung übersetzt.

```
t                    --->   IF sy=t THEN
                              Lesen
                            ELSE Fehler END;
```

2. Jedes Hilfssymbol h wird in einen Aufruf der zugehörigen Parserprozedur übersetzt.

```
h                    --->   h;
```

3. Unmittelbar aufeinanderfolgende Syntaxausdrücke werden in eine Anweisungsfolge übersetzt.

```
a1 a2 ... an         --->   P[a1]; P[a2]; ...; P[an]
```

4. Alternativen sind in CASE-Anweisungen zu übersetzen.

```
a1 | a2 |     | an    --->   CASE sy OF
                               fk[a1]: P[a1];
                             | fk[a2]: P[a2];
                              ...
                             | fk[an]: P[an];
                             ELSE Fehler END;
```

5. Konstruktionen mit eckigen Klammern werden in CASE- oder bedingte Anweisungen übersetzt.

```
[ a ]                --->   CASE sy OF
                               fk[a]: P[a];
                             ELSE END;
```

```
[ a ]                ---->   IF or[a] THEN
                               P[a];
                             END;
```

6. Geschweifte Klammern führen auf eine WHILE-Anweisung oder auf eine LOOP-CASE-Kombination.

```
{ a }                ---->   WHILE or[a] DO
                               P[a];
                             END;

{ a }                ---->   LOOP
                               CASE sy OF
                                 fk[a]: P[a];
                               ELSE EXIT END;
                             END;
```

7. Runde Klammern haben keine besondere Wirkung.

```
( a )                ---->   P[a];
```

Wirth gibt in [Wirt77] für häufig anzutreffende Fälle zusätzliche Regeln an, in denen mögliche Vereinfachungen des generierten Programmtexts bereits berücksichtigt sind. Ich füge noch zwei hinzu.

8. Führende Terminalsymbole t1, t2, ..., tn bei den einzelnen Varianten einer Alternative

```
t1 a1 | t2 a2 |     | tn
                     ---->   CASE sy OF
                               t1: Lesen; P[a1];
                             | t2: Lesen; P[a2];
                              ...
                             | tn: Lesen; P[an];
                             ELSE Fehler END;
```

9. Führendes Terminalsymbol t in geschweifter Klammer

```
{ t a }              ---->   WHILE sy=t DO
                               Lesen; P[a];
                             END;
```

10. Wiederholungen mit abschliessendem Terminalsymbol t

```
{ a } t              ---->   LOOP
                               CASE sy OF
                                 fk[a]: P[a];
                               | t: Lesen; EXIT;
                               ELSE Fehler END;
                             END;
```

11. Listenkonstruktionen mit trennendem Terminalsymbol t

```
a { t a }          --->    LOOP
                             P[a];
                             IF sy#t THEN EXIT END;
                             Lesen;
                           END;
```

Exerzieren wir die Übersetzung an einem einfachen, aber trotzdem wirklichkeitsnahen Beispiel durch! Eine Messreihe soll verarbeitet werden. Die Struktur der Messdaten sei verbindlich durch folgende EBNF-Syntax mit dem Startsymbol 'messreihe' beschrieben:

```
messreihe = {eintrag neue_zeile} "#"
eintrag = "a" zahl
        | "b" paar
        | "c" tripel .
zahl = " " {" "} ziffer {ziffer}
paar = zahl zahl .
tripel = zahl zahl zahl .
neue_zeile = "36C" .
ziffer = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Für jedes Hilfssymbol entsteht eine Prozedur. Zur Verringerung der Prozeduranzahl sollte man alle Hilfssymbole, die nur "selten" benutzt werden und deren Regel "kurz" ist, beseitigen: Einsetzen der rechten Seiten der zu den fraglichen Hilfssymbolen gehörenden Regeln an den entsprechenden Stellen. Ich mache das für 'paar', 'tripel' sowie 'neue_zeile' und erhalte:

```
messreihe = {eintrag "36C"} "#"
eintrag = "a" zahl
        | "b" zahl zahl
        | "c" zahl zahl zahl .
zahl = " " {" "} ziffer {ziffer} .
ziffer = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Für die Behandlung der Klammerkonstruktion in 'messreihe' ist Übersetzungsregel 10 zuständig. Die Menge FIRST(eintrag "36C") besteht aus den drei Symbolen "a", "b" und "c". Also ist

```
fk[eintrag "36C"]  =  "a","b","c".
```

Die aus der rechten Seite der Syntaxregel 'messreihe' entstehende LOOP-CASE-Kombination hat demnach folgendes Aussehen:

```
P[{eintrag "36C"} "#"]  =  LOOP
                             CASE sy OF
                               "a","b","c": P[eintrag "36C"];
                             | "#": Lesen; EXIT;
                             ELSE Fehler END;
```

Nun wären P[eintrag] nach Regel 2 und P["36C"] nach Regel 1 zu bestimmen.

```
P[eintrag]  =  eintrag;     p["36C"]  =  IF sy="36C" THEN
                                           Lesen;
                                         ELSE Fehler END;
```

Wenn wir anschliessend beides gemäss Regel 3 zusammenfügen und das Resultat in die CASE-Anweisung einsetzen, dann erhalten wir die Zeilen 3 bis 11 der Prozedur 'messreihe'.

```
 1 PROCEDURE messreihe;
 2 BEGIN
 3   LOOP
 4     CASE sy OF
 5       "a","b","c": eintrag;
 6                    IF sy=36C THEN
 7                      Lesen;
 8                    ELSE Fehler END;
 9     | "#": Lesen; EXIT;
10     ELSE Fehler END;
11   END;
12 END messreihe;
```

Das Innenleben der Prozedur 'eintrag' entsteht im wesentlichen durch Anwenden der Regel 8. In den einzelnen Varianten wurden die Regeln 2 und 3 benutzt.

```
 1 PROCEDURE eintrag;
 2 BEGIN
 3   CASE sy OF
 4     "a": Lesen; zahl;
 5   | "b": Lesen; zahl; zahl;
 6   | "c": Lesen; zahl; zahl; zahl;
 7   ELSE Fehler END;
 8 END eintrag;
```

Für die zweimalige Anwendung von Regel 6 bei 'zahl' benötigt man die Bedingungen or[" "] und or[ziffer]. Formales Herangehen führt

auf

```
or[" "]     =  sy=" "       und
or[ziffer]  =  (sy="0")OR(sy="1")OR...OR(sy="9").
```

Aus der Sicht eines denkenden Programmierers ist die zweite Bedingung ausgesprochen ungeschickt formuliert. Also ändern wir sie gleich ab.

```
or[ziffer]  =  (sy>="0")AND(sy<="9")
```

Die Bedeutung hat sich nicht geändert. Nach dieser Vorüberlegung kann man die Prozedur 'zahl' sofort hinschreiben.

```
 1 PROCEDURE zahl;
 2 BEGIN
 3   IF sy=" " THEN
 4     Lesen;
 5   ELSE Fehler END;
 6   WHILE sy=" " DO
 7     Lesen;
 8   END;
 9   ziffer;
10   WHILE (sy>="0")AND(sy<="9") DO
11     ziffer;
12   END;
13 END zahl;
```

Beim Aufschreiben von Prozedur 'ziffer' greift man am besten auf Regel 8 zurück. Als führende Terminalsymbole treten die einzelnen Ziffern auf. Die Syntaxausdrücke a1, a2, ..., a10 sind leer. Also sind auch alle P[ai] (i=1, 2, ..., 10) Leeranweisungen. Und daraus ergibt sich, dass in allen 10 Fällen dieselbe Aktion auszuführen ist: 'Lesen'. Ich fasse deshalb alles zu einem Fall zusammen.

```
1 PROCEDURE ziffer;
2 BEGIN
3   CASE sy OF
4     "0".."9": Lesen;
5   ELSE Fehler END;
6 END ziffer;
```

Nehmen wir abschliessend das Zusammenspiel von 'zahl' und 'ziffer' kritisch unter die Lupe! Voraussetzung für das Erreichen von Zeile 11 in 'zahl' ist, dass 'sy' eine Ziffer enthält. Der innerhalb von 'ziffer' stattfindende Test, ob es sich beim Wert von 'sy' auch

wirklich um eine Ziffer handelt, ist dann überflüssig. Es wird also sinnvoll sein, das zunächst ziemlich formal erstellte Programm nachzubessern. Ich würde die einzige in 'ziffer' erfolgende Aktion 'Lesen' gleich in Zeile 11 von 'zahl' an Stelle des Aufrufs 'ziffer' notieren.

Zur Demonstration des "Einflechtens semantischer Aktionen" in den Parser stellen wir uns die Aufgabe, die Einträge in einer Messreihe zu zählen und für jeden Typ (a, b und c) die Anzahl auszuschreiben. Was muss getan werden? Im Programmblock führe ich drei Zählvariablen ein und setze sie auf den Anfangswert 0. Die INC-Operationen zum Abbuchen der einzelnen Einträge sind in den drei Fällen der CASE-Anweisung im Körper von 'eintrag' zu plazieren. Das Ausschreiben der Werte steht am Schluss des Hauptprogramms nach der Rückkehr aus 'messreihe'.

Programm 8.3 zeigt die fertige Lösung. Alles, was zu den semantischen Aktionen gehört, ist durch Fettdruck hervorgehoben. Das Programm enthält darüber hinaus Veränderungen bez. der Fehlerbehandlung und liefert ein Protokoll der eingelesenen Daten.

```
 1 MODULE MessReihenVerarbeitung;
 2 (*-------------------------------------------------------------
 3   Beispiel zum syntaxorientierten Programmentwurf:
 4             ein um Zaehloperationen aufgeruesteter Parser
 5 -------------------------------------------------------------*)
 6 FROM InOut IMPORT
 7   Read,Write,WriteString,WriteLn,WriteInt,OpenInput,Done;
 8 (*-------------------------------------------------------------*)
 9 VAR   sy: CHAR;
10
11 PROCEDURE Lesen;
12 BEGIN
13   Read(sy); Write(sy);
14 END Lesen;
15
16 PROCEDURE Fehler(txt: ARRAY OF CHAR);
17 BEGIN
18   WriteString("<-- "); WriteString(txt); WriteLn;
19 END Fehler;
```

```
20
21 PROCEDURE Ueberlesen(StabilisierungsSymbol: CHAR);
22 BEGIN
23   REPEAT Lesen;
24     IF NOT Done THEN
25       WriteString("<--- File-Ende"); WriteLn; HALT;
26     END;
27   UNTIL sy=StabilisierungsSymbol;
28   WriteString("+++>"); WriteLn; Lesen;
29 END Ueberlesen;
30
31 PROCEDURE messreihe;
32 BEGIN
33   LOOP
34     CASE sy OF
35       "a","b","c": eintrag;
36                    IF sy=36C THEN Lesen;
37                    ELSE Fehler("Zeilenende erwartet");
38                      Ueberlesen(36C);
39                    END;
40     | "#": Lesen; EXIT;
41     ELSE Fehler('Kennung oder "#" erwartet');
42       Ueberlesen(36C);
43     END;
44   END;
45 END messreihe;
46
47 PROCEDURE eintrag;
48 BEGIN
49   CASE sy OF
50     "a": Lesen; zahl; INC(aZaehler);
51   | "b": Lesen; zahl; zahl; INC(bZaehler);
52   | "c": Lesen; zahl; zahl; zahl; INC(cZaehler);
53   ELSE Fehler("Falsche Kennung") END;
54 END eintrag;
55
```

```
 56 PROCEDURE zahl;
 57 BEGIN
 58   IF sy=" " THEN Lesen;
 59   ELSE Fehler("Leerzeichen erwartet") END;
 60   WHILE sy=" " DO Lesen END;
 61   ziffer;
 62   WHILE (sy>="0")AND(sy<="9") DO Lesen END;
 63 END zahl;
 64
 65 PROCEDURE ziffer;
 66 BEGIN
 67   CASE sy OF
 68     "0".."9": Lesen;
 69   ELSE Fehler("Ziffer erwartet") END;
 70 END ziffer;
 71
 72 VAR
 73   aZaehler, bZaehler, cZaehler: INTEGER;
 74
 75 PROCEDURE ZaehlerAusschreiben;
 76   PROCEDURE S(txt: ARRAY OF CHAR; n: INTEGER);
 77   BEGIN WriteString(txt); WriteInt(n,6); WriteLn END S;
 78 BEGIN WriteLn;
 79   S(' Eintraege der Sorte "a":', aZaehler);
 80   S(' Eintraege der Sorte "b":', bZaehler);
 81   S(' Eintraege der Sorte "c":', cZaehler);
 82   S('Gesamtzahl der Eintraege:', aZaehler+bZaehler+cZaehler);
 83 END ZaehlerAusschreiben;
 84 (*----------------------------------------------------------*)
 85 BEGIN
 86   aZaehler:=0; bZaehler:=0; cZaehler:=0;
 87   OpenInput("mess.dat");
 88   Lesen; messreihe; ZaehlerAusschreiben;
 89 END MessReihenVerarbeitung.
```

Programm 8.3. Ein Parser mit nachträglich eingefügten semantischen Aktionen

Das Protokoll entsteht, indem jedes eingelesene Zeichen sofort wieder ausgeschrieben wird (Prozedur 'Lesen'). Diese mitlaufende Kopie der Eingabedaten ist eine gute Hilfe bei der Fehlersuche:

> Das letzte protokollierte Zeichen wird dasjenige sein, bei dem die Syntaxanalyse nicht mehr weiterkam.

Wenn man die Prozedur 'Fehler' mit einem Parameter 'txt' ausrüstet, der eine Fehlermeldung enthält, und diese Zeichenkette vor dem Abbruch ausschreibt, dann sind Syntaxfehler in den Eingabedaten nach und nach relativ leicht behebbar. Man startet das Programm, kommt bis zum ersten Fehler, repariert ihn nach dem Abbruch, startet erneut und findet den nächsten Fehler.

Wünschenswert ist eine Fehlerstabilisierung: Nach dem Erkennen eines Fehlers wird die Analyse nicht abgebrochen, sondern an einer "geeigneten" Stelle der Eingabedaten fortgesetzt. Auf diese Weise ist es möglich, bei einem Lauf des Programms gleich mehrere Fehler zu finden.

Also habe ich den Ruf der Prozedur 'HALT' in 'Fehler' gestrichen und die Prozedur 'Ueberlesen' (Zeilen 21 bis 29) hinzugefügt. Sie liest und "verschluckt" Eingabesymbole, bis sie auf ein Symbol getroffen ist, hinter dem die Analyse fortgesetzt werden soll (Parameter 'StabilisierungsSymbol'). Der Programmierer kann die Analyse nach einem Fehler einfach weiterlaufen lassen, oder er ruft 'Ueberlesen' mit einem geeigneten Parameter auf. Bei unserem Beispiel bietet es sich in zwei Situationen an, bis zum Zeilenende alles zu ignorieren, und in der neuen Zeile wieder mit einem vernünftigen 'eintrag' zu rechnen (s. Zeilen 38 und 42). Die Notbremse 'HALT' wird nur dann gezogen, wenn 'Ueberlesen' auf das File-Ende stösst.

Wenn dem Programm 'MessReihenVerarbeitung' ein File 'mess.dat' vorlegt wird, das aus den Zeilen

```
a              1
b12          123
d 123 1234 12345
aa
b 1 2 3
c 1 2 3
#
```

besteht und offenbar einige Syntaxfehler enthält, dann wird man am

Bildschirm die folgenden Ausschriften beobachten können:

```
a              1
b1<-- Leerzeichen erwartet
2          123
d<-- Kennung oder "#" erwartet
 123 1234 12345
+++>
aa<-- Leerzeichen erwartet
<-- Ziffer erwartet
<-- Zeilenende erwartet

+++>
b 1 2 <-- Zeilenende erwartet
3
+++>
c 1 2 3
#
 Eintraege der Sorte "a":     2
 Eintraege der Sorte "b":     2
 Eintraege der Sorte "c":     1
Gesamtzahl der Eintraege:     5
```

Eine Interpretation dieses Protokolls dürfte mit Hilfe des Quelltexts von Programm 8.3 nicht schwierig sein.
Tiefgründigere Untersuchungen und Erfahrungsberichte zur Frage der Fehlerstabilisierung bei Parsern der beschriebenen Art findet man z. B. in [Amma75], [Amma78] und [Polz79]. Oft ist es nicht ausreichend, ein einzelnes Stabilisierungssymbol zu betrachten.

Ein zweites Parserbeispiel befasst sich mit der Darstellung der "Ist-Teil-von"-Beziehung. Ein Auto besteht aus Karosserie, Rädern und Motor. Teile eines Buches sind Einband und Blätter. Zu den Gerätschaften auf einem Frühstückstisch gehören Geschirr, Blumenvase und Besteck.
Wenn solche Teil/Ganzes-Beziehungen in einem Computer gespeichert und verarbeitet werden sollen, dann muss man Wege finden, um sie zu beschreiben und einzulesen. Eine Möglichkeit besteht darin, die Dinge durch Aufschreiben ihres Namens darzustellen und ggf. eine Liste der Teile dahinter zu schreiben. Wenn ein Ding als nicht

weiter zerlegbar angesehen wird, wenn es also keine Teile hat, dann entfällt die Teileliste. Hierfür wäre folgende Syntax denkbar:

```
ding = name [teileliste] .
name = buchstabe {buchstabe} .
teileliste = "(" teil {"," teil} ")"
buchstabe = "a"|"b"|...|"z"|"A"|"B"|...|"Z"
```

Was soll als 'teil' zugelassen werden? Sicher ist 'name' eine vernünftige Lösung. Aber warum soll eigentlich ein Teil nicht auch Teile haben können? Zum Frühstücksgeschirr gehören Tasse, Untertasse, Teller und Kaffeekanne. Und die Kaffeekanne besteht auch noch aus Kanne und Deckel. Ich ergänze also die Syntax um die Regel

```
teil = ding .
```

Die bereits zweimal erwähnten Gerätschaften auf dem Frühstückstisch einer einzelnen Person können entsprechend dieser Syntax als

```
Fruehstuecksgeraete( Geschirr( Tasse
                             , Teller
                             , Untertasse
                             , Kaffeekanne(Kanne, Deckel)
                             )
                   , Blumenvase
                   , Besteck(Messer, Kaffeeloeffel)
                   )
```

beschrieben werden. Vor der Transformation der Syntaxregeln in Parserprozeduren setze ich zuerst die rechte Seite der Regel 'teil' in 'teileliste' ein und mache danach dasselbe mit 'teileliste' in 'ding'. Die "neue" Syntax hat drei Regeln:

```
ding = name ["(" ding {"," ding} ")"]
name = buchstabe {buchstabe} .
buchstabe = "a"|"b"|...|"z"|"A"|"B"|...|"Z" .
```

Der aus dieser Syntax abgeleitete Parser ist auf den folgenden zwei Seiten als **Programm 8.4** abgedruckt. Gemäss Übersetzungsregel 2 von Seite 116 kommen innerhalb der Prozedur 'ding' auf den Zeilen 40 und 41 Aufrufe von 'ding' vor: 'ding' ist eine rekursiv definierte Prozedur.

Um übersichtlich eingerückte Notationen (s. Beispiel Fruehstuecksgeraete) zu ermöglichen, lege ich fest, dass nur in der Regel 'name' die Terminalsymbole unmittelbar aufeinanderfolgen müssen. An allen anderen Stellen sind beliebig viele Leerzeichen und Zei-

lenwechsel erlaubt. Die Prozedur 'Lesen' (s. Zeilen 16 bis 19) liefert deshalb in 'sy' jeweils das nächste Zeichen, das kein Steuer- und kein Leerzeichen ist. 'Lesen1' liest einzelne Zeichen.

```
 1 MODULE TeileHierarchie;
 2 (*-------------------------------------------------------------
 3   Beispiel zum syntaxorientierten Programmentwurf:
 4            ein Parser fuer Teilehierarchien
 5 -------------------------------------------------------------*)
 6 FROM InOut IMPORT
 7   Read, Write, WriteString, WriteLn, OpenInput, Done;
 8 (*-------------------------------------------------------------*)
 9 VAR   sy: CHAR;
10
11 PROCEDURE Lesen1;
12 BEGIN
13   Read(sy); Write(sy);
14 END Lesen1;
15
16 PROCEDURE Lesen;
17 BEGIN
18   REPEAT Lesen1 UNTIL (sy>" ") OR NOT Done;
19 END Lesen;
20
21 PROCEDURE Fehler(txt: ARRAY OF CHAR);
22 BEGIN
23   WriteString("<-- "); WriteString(txt); WriteLn;
24 END Fehler;
25
26 PROCEDURE Ueberlesen(StabilisierungsSymbol: CHAR);
27 BEGIN
28   REPEAT Lesen;
29     IF NOT Done THEN
30       WriteString("<--- File-Ende"); WriteLn; HALT;
31     END;
32   UNTIL sy=StabilisierungsSymbol;
33   WriteString("+++>"); WriteLn; Lesen;
34 END Ueberlesen;
```

```
35
36 PROCEDURE ding;
37 BEGIN
38   name;
39   IF sy="(" THEN
40     Lesen; ding;
41     WHILE sy="," DO Lesen; ding END;
42     IF sy=")" THEN Lesen ELSE
43       Fehler('")" erwartet'); Ueberlesen(")");
44     END;
45   END;
46 END ding;
47
48 PROCEDURE name;
49 BEGIN
50   IF (sy>="a")AND(sy<="z") OR (sy>="A")AND(sy<="Z") THEN
51     Lesen1;
52   ELSE Fehler("Buchstabe erwartet") END;
53   WHILE (sy>="a")AND(sy<="z") OR (sy>="A")AND(sy<="Z") DO
54     Lesen1;
55   END;
56   IF sy<=" " THEN Lesen END;
57 END name;
58 (*-------------------------------------------------------------*)
59 BEGIN
60   OpenInput("teile.dat");
61   Lesen; ding;
62 END TeileHierarchie.
```

Programm 8.4. Ein rekursiver Parser

Parserprozeduren, die nach den in diesem Abschnitt angegebenen Transformationsvorschriften aus Syntaxbeschreibungen in EBNF abgeleitet werden, haben bei der Verarbeitung syntaktisch korrekter Eingabedaten folgende wichtige Eigenschaft:

> Beim Verlassen der Prozedur steht in 'sy' schon das nächste Terminalsymbol zur Verarbeitung bereit.

Diese Eigenschaft darf bei der Einführung von Fehlerbehandlungen

und semantischen Aktionen nicht verlorengehen. Aus diesem Grund musste z. B. Zeile 56 in 'name' eingefügt werden.
In der Fachliteratur über Theorie und Technik des Compilerbaus wird die in diesem Abschnitt beschriebene Methode zur Syntaxanalyse als "Verfahren des rekursiven Abstiegs" bezeichnet: Mit Hilfe rekursiv definierter Prozeduren steigt man vom Satzsymbol der Grammatik bis hinab zu den Terminalsymbolen.
Nicht für jede EBNF-Grammatik kann ein nach dem Verfahren des rekursiven Abstiegs arbeitender Parser angegeben werden. Es ist nur dann möglich, wenn die Grammatik der sog. LL(1)-Bedingung genügt. Das vorgestellte Parserkonstruktionsverfahren funktioniert also nur bei Grammatiken, die diese Bedingung erfüllen. Zum Formulieren der LL(1)-Bedingung benötige ich für Syntaxausdrücke a mit eckigen oder geschweiften Klammern die Menge FOLLOW(a).
Gegeben sei eine EBNF-Grammatik G=(T,H,R,s). Dabei sind T und H die Mengen der Terminal- bzw. Hilfssymbole; s ist das Startsymbol, und R ist die Menge der in EBNF formulierten Syntaxregeln (vgl. Abschn. 4.1). a sei ein innerhalb der rechten Seite einer Regel aus R auftretender Syntaxausdruck.

Mit FOLLOW(a) wird die Menge aller jener Terminalsymbole bezeichnet, die unmittelbar hinter einer aus a (gemäss Punkt 1 bis 7 von Abschn. 4.1) ableitbaren Symbolfolge stehen können.

Eine Grammatik G genügt der LL(1)-Bedingung, wenn folgende Aussagen wahr sind:

1. Für jede in R auftretende Alternative a1 | a2 | ... | an sind die Mengen FIRST(ai) (i=1,2,...,n) paarweise durchschnittsfremd. Das heisst, keine zwei der Mengen enthalten ein gemeinsames Element.
2. Für jedes Auftreten einer Konstruktion der Gestalt [a] oder {a} in R gilt, dass die Mengen FOLLOW([a]) und FIRST(a) bzw. FOLLOW({a}) und FIRST(a) keine gemeinsamen Elemente enthalten.

Die erste Bedingung sichert, dass an Hand des ersten Terminalsymbols einer Alternative entschieden werden kann, welcher Zweig gemeint ist. Das Verletzen dieser Bedingung schlägt auf die aus der Alternativkonstruktion abgeleitete CASE-Anweisung durch. Die in den Fallkennzeichen auftretenden Konstanten sind nicht mehr eindeutig, und der Compiler meldet "case label defined twice".
Die zweite Bedingung sichert, dass durch Betrachten des aktuellen

Terminalsymbols entschieden werden kann, ob die eckig oder geschweift eingeklammerte Konstruktion kommt oder ob es dahinter weitergeht. Wenn die Bedingung verletzt ist, dann sind die nachfolgenden Konstruktionen ganz oder teilweise "unerreichbar". Der Parser beschreitet immer den durch den Klammerinhalt beschriebenen Weg. Bei dem durch Transformationsregel 10 beschriebenen Spezialfall wirken sich Verstösse direkt im zugehörigen Programmstück aus. Die Fallkennzeichen innerhalb der entsprechenden CASE-Anweisung sind nicht eindeutig.

Die zunächst abschreckend abstrakt wirkende LL(1)-Bedingung hat also eine recht anschauliche Interpretation, wenn man den aus der Grammatik abgeleiteten Parser betrachtet. Sehr angenehm ist, dass die Prüfung der LL(1)-Bedingung weitgehend von dem den Parser übersetzenden Compiler erledigt wird.

Ich vertrete bez. des Entwurfs von Datenstrukturen oder Spezialsprachen, für die dann auch noch ein Parser und daraus abgeleitete Programme gebraucht werden, die Auffassung, dass man nichts entwerfen sollte, das nicht mit einer LL(1)-Grammatik beschreibbar ist. Hierin liegt offenbar keine gewaltige Einschränkung. Die Syntax von PASCAL und auch die Syntax von MODULA-2 sind (fast) LL(1)-gerecht.

9. Datentypen

Neben den vordefinierten Standardtypen (s. Abschn. 6) bietet die Sprache MODULA-2 dem Programmierer Möglichkeiten zur Definition eigener problembezogener Datentypen an.

1. Aufzählungstypen und Teilbereichstypen ergänzen das Reservoir der diskreten Datentypen INTEGER, CARDINAL, CHAR und BOOLEAN in natürlicher Weise.
2. RECORD- und ARRAY-Typen fassen Komponenten von verschiedenem oder gleichem Typ zusammen und stellen Mittel für den Zugriff auf die Strukturkomponenten bereit.
3. SET-Typen modellieren "kleine" Mengen und machen Operationen wie Durchschnitt, Vereinigung und Test auf Vorhandensein eines bestimmten Elements verfügbar.
4. PROCEDURE-Typen gestatten die Einführung von Variablen und Parametern, deren Wert Prozeduren mit vom Programmierer vorgegebener abstrakter Signatur sind.
5. POINTER-Typen erschliessen die Welt der dynamischen Datenstrukturen, für deren Aufbau erst während der Programmabbarbeitung gezielt Speicherplatz angefordert wird.

Die Datentypen werden in der MODULA-2-Syntax durch das Hilfssymbol 'typ' repräsentiert. Abschnitt 5.2 enthält eine unvollständige Regel dafür.

```
$    typ = q_name | diskreter_typ | record_typ | array_typ
$        | set_typ | prozedur_typ | pointer_typ .
```

Ein 'typ' ist die Beschreibung eines Datentyps. Eine Typvereinbarung führt einen Typnamen ein.

```
$    typvereinbarung = bezeichner "=" typ .
```

Auf den durch 'typ' beschriebenen Datentyp kann fortan mittels des Bezeichners Bezug genommen werden. Typvereinbarungen gehören zu den am Anfang von Blöcken auftretenden Vereinbarungen (s. 'block' im Abschn. 5.1). Ich ergänze die dort angegebene Regel 'vereinbarung' entsprechend und füge auch die im Abschnitt 13.1 kurz zu erwähnen-

den Vereinbarungen lokaler Moduln hinzu. Die Regel ist damit endgültig vervollständigt.

```
$    vereinbarung = TYPE {typvereinbarung ";"}
$                 | VAR {variablenvereinbarung ";"}
$                 | CONST {konstantenvereinbarung ";" }
$                 | prozedurvereinbarung ";"
$                 | modulvereinbarung ";"
```

Typvereinbarungen können die Klarheit und Lesbarkeit von Programmen verbessern, wenn entsprechende Typnamen gewählt werden. Solange es nur um Variablen von einem nutzerdefinierten Typ geht, könnte auf Typvereinbarungen verzichtet werden (s. Syntaxregel 'variablenvereinbarung' im Abschn. 5.2). Aber bei Parametern benötigt man als Typangabe einen Bezeichner (s. Syntax von 'signatur' usw. im Abschn. 5.2). Die erste Variante in der Regel 'typ' zeigt, dass es möglich ist, für Typen neue Namen (bisweilen Aliasnamen genannt) einzuführen.
Zur Beschreibung eines Datentyps gehören die Darstellung seiner Wertmenge und die Angabe der mit diesen Werten möglichen Operationen. Die unterschiedlichen Varianten für 'typ' beschreiben bei MODULA-2 explizit die Wertmengen. Die möglichen Operationen stehen für jede dieser Varianten fest. Sie sind gültig für alle RECORD-Typen, für alle SET-Typen usw.

9.1. Aufzählungstypen und Teilbereiche

Die in den Abschnitten 6.1 und 6.2 sowie 6.3 und 6.4 behandelten Standardtypen INTEGER, LONGINT, CARDINAL, LONGCARD, CHAR und BOOLEAN beschreiben endliche geordnete Wertebereiche. Die Werte liegen nicht "dicht"; die Wertebereiche sind "diskret": Für jeden Wert (abgesehen vom letzten bzw. ersten) gibt es einen eindeutig definierten Nachfolger und einen eindeutig definierten Vorgänger.
Bemerkung: Im Gegensatz dazu liegen die mathematischen reellen Zahlen und auch die mathematischen rationalen Zahlen dicht: Zwischen je zwei verschiedenen reellen bzw. rationalen Zahlen gibt es unendlich viele verschiedene reelle bzw. rationale Zahlen. Das

Attribut "mathematisch" habe ich hier zur deutlichen Abgrenzung von den REAL- und LONGREAL-Computerzahlen (s. Abschn. 11) benutzt.
In MODULA-2 kann der Programmierer zwei Arten eigener diskreter Datentypen einführen:

- Aufzählungstypen definieren völlig neue Wertemengen.
- Teilbereiche beschreiben Intervalle innerhalb eines diskreten "Vatertyps".

Beginnen wir mit den Aufzählungstypen!

```
$    diskreter_typ = aufzaehlungstyp | teilbereichstyp .
$    aufzaehlungstyp = "(" bezeichnerliste ")"
```

In einem 'aufzaehlungstyp' werden die Bezeichner der Typelemente in aufsteigender Ordnung notiert. Das erste Element hat die Ordnungszahl 0. Zum Beispiel führt die Zeile

```
(sehrgut, gut, befriedigend, genuegend, ungenuegend)
```

fünf Konstantenbezeichner ein und beschreibt einen Typ, dessen Werte die Konstanten 'sehrgut', ..., 'ungenuegend' sind. Gleichzeitig ist folgende Ungleichungskette festgelegt:

```
sehrgut < gut < befriedigend < genuegend < ungenuegend.
```

Mit Werten aus Aufzählungstypen ist keinerlei Arithmetik möglich. Nur die Vergleichsoperationen

```
<  <=  >  >=  =  <>  #
```

sind definiert. Die Standardprozeduren INC und DEC können zur Veränderung des Werts von Variablen v verwendet werden, deren Typ ein Aufzählungstyp ist. Sei i ganzzahlig; dann bewirken

INC(v), INC(v,i)	den Übergang zum Nachfolger bzw. i-ten Nachfolger des Werts von v und
DEC(v), DEC(v,i)	den Übergang zum Vorgänger bzw. i-ten Vorgänger des Werts von v.

Folgende Standardfunktionen sind anwendbar:

MIN(T)	kleinstes Element des Typs T
MAX(T)	grösstes Element des Typs T
ORD(x)	Ordnungszahl der Konstanten x (CARDINAL)
VAL(T,c)	Konstante mit der Ordnungszahl c in T.

Bei VAL(T,c) muss c vom Typ CARDINAL sein. Für Werte x aus einem Aufzählungstyp T gilt die Gleichung

```
VAL( T, ORD(x) ) = x.
```

Aufzählungstypen sollten immer dort verwendet werden, wo Zustände oder Kennzeichen für irgendwelche Varianten oder auch Kürzel für verbale Bewertungen zu modellieren sind. Die Gemeinsamkeit dieser Dinge ist, dass man unterscheidbare Werte braucht, die nicht sinnvoll arithmetischen Verknüpfungen unterzogen werden können. Die Ordnung ist oft wertvoll; bisweilen spielt auch sie keine Rolle. In dem angegebenen Beispiel mit den Verbalzensuren ist die Ordnung wichtig: "kleiner" bedeutet "besser". Arithmetik - wie etwa die Berechnung irgendwelcher Durchschnitte - hat keinen rechten Sinn.
Weitere Beispiele für diese Sorte nutzerdefinierter Typen werden vor allem im Zusammenhang mit Variantenteilen bei RECORD-Typen (s. Abschnitt 9.2) auftreten. Der Standardtyp BOOLEAN kann als Aufzählungstyp

```
TYPE
  BOOLEAN = (FALSE, TRUE);
```

mit einigen zusätzlichen Operationen verstanden werden. Oft sind Aufzählungstypen ein Merkmal klarer, gut lesbarer Programme. Aber Niklaus Wirth, der Schöpfer von MODULA-2, warnt [Wirt87]. Er hat "bei einer wachsenden Anzahl von Programmen beobachtet, dass die kritiklose Verwendung von Aufzählungstypen zu einem pompösen Stil führte, der nicht zur Klarheit der Programme, sondern zu deren Weitschweifigkeit beitrug".
Während Aufzählungstypen neue Werte einführen, beschreiben die Teilbereichstypen zweckgebundene Einschränkungen des Wertebereichs bereits existierender diskreter Typen (Basistyp). Die ausführbaren Operationen erbt ein Teilbereichstyp von seinem Basistyp.

```
$    teilbereichstyp = [q_name] teilbereich.
   Der 'q_name' muss einen Aufzählungstyp, einen INTEGER- oder
   CARDINAL-Typ oder einen der Typen CHAR und BOOLEAN beschreiben.
$    teilbereich = "[" untere_grenze ".." obere_grenze "]"
$    untere_grenze = konstantenausdruck .
$    obere_grenze = konstantenausdruck .
```

Untere und obere Grenze müssen denselben Typ T haben. Wenn ein Typname ('q_name') angegeben ist, dann muss das der Typ der Grenzen sein. T heisst Basistyp des Teilbereichstyps. Der Wert der unteren Grenze darf nicht grösser sein als der Wert der oberen Grenze. Ein

Teilbereichstyp

```
[UG..OG]
```

beschreibt die Menge der Werte x seines Basistyps, für die gilt

```
UG <= x <= OG.
```

Als Basistyp eines Teilbereichs sind die INTEGER- und die CARDINAL-Typen sowie Aufzählungstypen und CHAR bzw. BOOLEAN zulässig.
Bei Wertzuweisungen an Variablen oder ARRAY- bzw. RECORD-Komponenten (s. Abschn. 9.2), deren Typ ein Teilbereichstyp ist, wird vor der Zuweisung geprüft, ob der fragliche Wert auch wirklich zwischen unterer und oberer Grenze liegt. Wenn das nicht der Fall ist, gibt es einen Fehlerabbruch des Programms wegen

```
value out of range.
```

Durch die Einführung einer Variablen

```
VAR
  Tag: [1..31];
```

veranlasse ich den Compiler, Sorge für die Einhaltung der Bedingung

```
1 <= Tag <= 31
```

zu tragen. An allen Stellen, wo der Variablen 'Tag' ein Wert zugewiesen wird, erzeugt der Compiler zusätzliche Testbefehle. Die Laufanweisung

```
FOR Tag:=3 TO 40 DO WriteInt(Tag, 3) END;
```

scheitert nach dem Ausschreiben des Wertes 31. Die Sinnfälligkeit dieser automatisch erzeugten Teilbereichstests ist in letzter Zeit umstritten, da der Programmierer keine Möglichkeit hat, in seinem Programm auf Verletzungen der Beschränkung zu reagieren. Der hauptsächliche Einsatzfall von Teilbereichstypen dürfte die Verwendung als Indextyp in ARRAY-Strukturen (s. Abschn. 9.2.2) sein.
Bemerkung für PASCAL-Kenner: Bei MODULA-2 gehören die eckigen Klammern zur Notation des Teilbereichstyps, bei PASCAL nicht.
Wenn bei numerischen Teilbereichen kein Typ vor der öffnenden eckigen Klammer angegeben ist, dann wird der Basistyp nach folgenden Regeln festgelegt:

1. Wenn die untere Grenze negativ ist, dann ist der Basistyp INTEGER oder LONGINT, andernfalls CARDINAL oder LONGCARD.
2. Wenn das gesamte Intervall in INTEGER bzw. CARDINAL liegt, dann ist das der Basistyp, ansonsten ist es der entsprechende LONG-Typ.

Auf Grund dieser Regeln gehört zur Beispielvariablen 'Tag' der

Basistyp CARDINAL. Arithmetische Verknüpfungen mit INTEGER-Variablen sind deshalb ausgeschlossen. Wenn ich erzwingen möchte, dass INTEGER der Basistyp von 'Tag' ist, dann brauche ich nur eine andere Variablenvereinbarung zu notieren.

```
VAR
  Tag: INTEGER[1..31];
```

Dieses Beispiel zeigt, wo die Typangabe vor dem Teilbereich sinnvoll ist. Diese Möglichkeit wurde erst mit den Sprachmodifikationen [Wirt84] eingebracht und ist in älteren MODULA-2-Systemen noch nicht verfügbar.

9.2. Strukturierte Typen

MODULA-2 kennt zwei Methoden zur Bildung strukturierter Werte:

1. In RECORD-Strukturen werden Komponenten, die von unterschiedlichem Typ sein dürfen, zu einem Aggregat zusammengefasst. Jede Komponente wird durch einen "Namen" identifiziert.
2. ARRAY-Strukturen sind Anreihungen von Komponenten desselben Typs. Die einzelnen Komponenten werden durch "Abzählen" identifiziert.

Beide Strukturierungsmethoden können beliebig miteinander kombiniert werden. Ein Beispiel dafür enthält schon der Programmodul 'Staedte' (Programm 2.2). Die Variable 'tabelle' ist eine ARRAY-Struktur, deren Komponenten RECORD-Strukturen der Sorte 'Eintrag' sind. Und das Datenfeld 'stadt' in 'Eintrag' ist wieder ein Array.

9.2.1. RECORD-Typen und WITH-Anweisungen

Die Komponenten in einem RECORD-Typ werden Datenfelder oder kurz Felder genannt.

```
$    record_typ = RECORD liste_von_feldlisten END .
$    liste_von_feldlisten = feldliste {";" feldliste} .
$    feldliste = [ bezeichnerliste ":" typ | variantenliste ]
```

Lassen wir die 'variantenliste zunächst beiseite!

Die Vereinbarungen

```
TYPE
  Monat = (jan,feb,mar,apr,mai,jun,jul,aug,sep,okt,nov,dez);
  Datum = RECORD
            t: [1..31]; m: Monat; j: [1900..2100];
          END;
VAR
  Geburtstag: Datum;
```

legen fest, dass die Variable 'Geburtstag' drei Datenfelder 't', 'm' und 'j' von recht unterschiedlichen Typen enthält. Werte von RECORD-Typen bezeichnet man oft auch als "Records" oder ins Deutsche übersetzt als "Datensätze".
Wie wird ein Zugriff auf das Datenfeld 'm' der Datensatzvariablen 'Geburtstag' notiert? Zur Erläuterung dieser Sache vervollständige ich die im Abschnitt 7.1 angegebene Syntaxregel 'bezugnahme'.

```
$    bezugnahme = q_name {selektor}
$    selektor = record_selektor
$             | array_selektor
$             | pointer_zugriff .
$    record_selektor = "." bezeichner .
```

Ein 'record_selektor' besteht aus dem Punkt und einem Datenfeldnamen. Der 'array_selektor' und der 'pointer_zugriff' werden in den Abschnitten 9.2.2 bzw. 12 behandelt. Die drei Wertzuweisungen

```
Geburtstag.t := 6;
Geburtstag.m := jan;
Geburtstag.j := 1982;
```

"füllen" alle Datenfelder von 'Geburtstag'. Datensatzkonstanten können in MODULA-2 nicht formuliert werden. Das heisst, es gibt keine Möglichkeit, die drei Ergibtanweisungen zu einer zusammenzufassen. Im Gegensatz dazu sind Wertzuweisungen zwischen Datensatzvariablen erlaubt. Wenn wir eine weitere Variable

```
VAR
  x: Datum;
```

vereinbaren, dann kann ihr mit einer einzigen Ergibtanweisung ein Wert zugewiesen werden.

```
x := Geburtstag;
```

Wenn man in einem Programmstück häufig Zugriffe auf Datenfelder desselben Datensatzes zu notieren hat, dann kann man mittels einer WITH-Anweisung den Datensatznamen "ausklammern".

$ with_anweisung = WITH bezugnahme DO anweisungsfolge END .
Die 'bezugnahme' muss auf einen Datensatz führen.

Innerhalb der 'anweisungsfolge' können die Datenfeldnamen aus dem Typ des durch die 'bezugnahme' angegebenen Datensatzes direkt benutzt werden. Sie bewirken den Zugriff auf die Datenfelder des angegebenen Datensatzes. Die WITH-Anweisung

```
WITH Geburtstag DO
  t:=6; m:=jan; j:=1982;
END;
```

leistet dasselbe wie die drei vorhin aufgeschriebenen Ergibtanweisungen.
Eine WITH-Anweisung ist mehr als nur syntaktischer Zucker zur Verkürzung der Schreibweise. Hier findet bei der Programmabarbeitung etwas statt, das grosse Ähnlichkeit mit Parametervermittlung und Bezugnahmen auf Referenzparameter bei Prozeduren hat:

Der zwischen WITH und DO notierte Datensatz wird beim "Betreten" der WITH-Anweisung bestimmt und fixiert. Alle durch blanke Datenfeldbezeichner notierten Zugriffe beziehen sich dann auf das beim Eintritt in die WITH-Anweisung ermittelte Objekt.

Diese Verfahrensweise wirkt sich besonders dann günstig auf die Abarbeitungsgeschwindigkeit aus, wenn die 'bezugnahme' zeitaufwendig ist, also z. B. mehrere ARRAY-Selektoren und Pointerzugriffe enthält. Der Aufwand wird nicht bei jedem Zugriff, sondern einmalig am Anfang der WITH-Anweisung betrieben.
RECORD-Typen, RECORD-Selektoren und WITH-Anweisungen machen Ergänzungen der im Abschnitt 5.4 behandelten Regeln bez. Eindeutigkeit und Gültigkeitsbereich von Bezeichnern notwendig.

1. Alle in einem RECORD-Typ eingeführten Datenfeldnamen - einschliesslich der in Variantenlisten für Daten- und Auswahlfelder notierten Namen - müssen voneinander verschieden sein.
2. Die Datenfeldnamen eines RECORD-Typs T sind nur in solchen RECORD-Selektoren und WITH-Anweisungen gültig, die sich auf Datensätze des Typs T beziehen.

Wichtige Konsequenzen aus Punkt 2 sind, dass erstens bei den Vereinbarungen

```
VAR
  Heute: Datum; t: BOOLEAN;
```

kein Bezeichnungskonflikt auftritt (die Datenfeldbezeichner sind auf der Ebene dieser Variablenvereinbarungen nicht gültig, also unbekannt) und dass zweitens die Boolesche Variable 't' innerhalb der Anweisungsfolge einer WITH-Anweisung

```
WITH Heute DO ... END;
```

nicht sichtbar ist. 't' bedeutet hier das Datenfeld 't' im Datensatz 'Heute'. Der Versuch, 't' den Wert 'TRUE' zuzuweisen:

```
WITH Heute DO ...;
  t:=TRUE;
END;
```

liefert demzufolge eine Quelltextfehlermeldung. Bei mir ist es

```
128: type incompatibility.
```

Bisweilen steht man als Programmierer vor der Situation, dass mehrere RECORD-Typen benötigt werden, die weitgehend übereinstimmen und sich nur in wenigen Datenfeldern unterscheiden. Betrachten wir ein Beispiel, das in den nächsten Abchnitten weitergeführt werden wird! Aufgabestellung und Lösungsweg orientieren sich an [Knau87] und dem dort publizierten PROLOG-Programm.

Beim Programmieren eines kleinen Expertensystems zur Diagnose technischer Geräte besteht eine Teilaufgabe in der Speicherung von "Symptomen", die während des Diagnoseprozesses systematisch erfragt werden müssen. Zu jedem Symptom gehört ein Stück Text. Es handelt sich dabei um die dem Bediener zu stellende Frage oder um die Bezeichnung einer rechnergestützt ablaufenden Messwerterfassung. Auch sollte man sich merken, ob ein Symptom schon erfragt wurde und wie das Ergebnis war. So können Wiederholungen derselben Frage vermieden werden. Es ist sinnvoll, drei Symptomarten zu unterscheiden:

1. Logische Symptome
   Eine Ja/Nein-Frage ist zu beantworten. (Leuchtet die Lampe 7 nach Betätigen von Schalter 12?)
2. Numerische Symptome
   Es muss ein Wert ermittelt und angegeben werden. Das Symptom

gilt als erfüllt, wenn der Wert in einem speziellen Intervall liegt. (Wie hoch ist die Netzspannung? Werte zwischen 210 und 230 Volt sind akzeptabel.)

3. Prozedurale Symptome
   Abarbeitung einer Prozedur, die on-line irgendwelche Messwerte abfragt und mitteilt, ob sie "zufrieden" ist oder nicht.

Spezifisch für logische Symptome wird ein Boolescher Wert sein, der anzeigt, ob mit "Ja" oder mit "Nein" geantwortet wurde. Zu numerischen Symptomen gehören die Grenzen des Akzeptanzintervalls und der wirklich mitgeteilte Wert. Bei prozeduralen Symptomen bietet sich ein prozedurwertiges Datenfeld in dem entsprechenden Datensatz an. Das angemessene programmiersprachliche Ausdrucksmittel für eine die Symptome beschreibende Datenstruktur ist ein RECORD-Typ mit einer Variantenliste.

```
$    variantenliste = CASE [bezeichner] ":" q_name OF
$                       variante {"|" variante}
$                     [ ELSE liste_von_feldlisten ]
$                     END .
```

Der 'q_name' muss sich auf einen Aufzählungstyp, einen INTEGER- oder CARDINAL-Typ bzw. auf CHAR oder BOOLEAN oder einen Teilbereichstyp beziehen.

```
$    variante =
$        [liste_von_fallkennzeichen ":" liste_von_feldlisten]
```

Aus der Diskussion über die Speicherung von Symptomen habe ich folgende Typvereinbarungen abgeleitet:

```
TYPE
  SymptomArten = (logisch, numerisch, prozedural);
  Symptom = RECORD
              txt: Kette;
              gesetzt: BOOLEAN;
              CASE art: SymptomArten OF
                logisch   : lwert: BOOLEAN;
              | numerisch : nmin,nmax,nwert: INTEGER;
              | prozedural: pwert: TestProzedur;
              END;
            END;
```

Die Vereinbarungen der Typbezeichner 'Kette' und 'TestProzedur' gebe ich in den Abschnitten über ARRAY- bzw. PROCEDURE-Typen an. Wegen der zweiten den Gültigkeitsbereich von Bezeichnern betreffenden Regel (s. Abschn. 5.4) müssen sie im Programmtext vor der Vereinbarung des Typs 'Symptom' stehen.
Die Datenfelder 'txt', 'gesetzt' und 'art' sind der allen drei Varianten von 'Symptom' gemeinsame Teil. 'art' ist einerseits ein ganz gewöhnliches Datenfeld. Andererseits spielt es die Rolle eines "Auswahlfeldes": Am Wert von 'art' ist ablesbar, welche Variante vorliegt.
Jede 'variante' wird mit den ihr entsprechenden Werten des Auswahlfeldes gekennzeichnet. Die 'liste_von_fallkennzeichen' ist bereits von der CASE-Anweisung (s. Abschn. 8.2) bekannt. In den zu einer 'variantenliste' gehörenden 'fallkennzeichen' darf kein Wert mehrmals vorkommen. Der Typ aller in den 'fallkennzeichen' auftretenden Konstantenausdrücke muss kompatibel zum Typ des Auswahlfeldes ('q_name') sein.
Logische Symptome haben das spezielle Datenfeld 'lwert'. Bei numerischen Symptomen sind an dieser Stelle drei INTEGER-Datenfelder vorhanden: 'nmin', 'nmax' und 'nwert'. In der dritten Variante gibt es dort die prozedurwertige Komponente 'pwert'.
Für nicht in den 'fallkennzeichen' auftretende Werte des Auswahlfeldes ist die ELSE-Variante zuständig. Die Speicherplätze der einzelnen Varianten - einschliesslich der ELSE-Variante - sind nicht hintereinander angeordnet, sondern beginnen jeweils an derselben Stelle: Sie "überlagern" sich. Der für eine 'variantenliste' benötigte Speicherplatz ist gleich dem Speicherplatzbedarf der längsten Variante. Wie den Syntaxregeln zu entnehmen ist, können in einem RECORD-Typ mehrere 'variantenlisten' nacheinander und auch verschachtelt auftreten.
Die nach meinem Geschmack unschöne Doppelverwendung des Schlüsselwortes CASE in MODULA-2 lässt sich vielleicht dadurch motivieren, dass bei der Verarbeitung von Datensätzen mit Varianten häufig CASE-Anweisungen verwendet werden: Abhängig vom Wert des Auswahlfeldes wird die jeweils passende Anweisungsfolge gewählt.
Die Aussage, dass am Wert des Auswahlfeldes ablesbar sei, welche Variante vorliegt, bedarf eines kritischen Kommentars. Mir ist keine Implementation von MODULA-2 bekannt, die beim Zugriff auf

Datenfelder einer Variante prüft, ob der aktuelle Wert des Auswahlfeldes dazu "passt". Das Zusammenspiel von Auswahlfeld und Varianten läuft demnach auf eine von den Programmierern einzuhaltende Konvention hinaus. Die beschriebene Verwendung von CASE-Anweisungen ist eine gute Methode.

Wenn von der Möglichkeit Gebrauch gemacht wird, den Bezeichner für das Auswahlfeld in der 'variantenliste' wegzulassen (s. eckige Klammern um 'bezeichner'), dann benötigt das Auswahlfeld keinen Speicherplatz. Es existiert nicht. Und man kann es deshalb auch nicht abtesten.

Ich verwende Datensätze ohne Auswahlfeld nur dann, wenn ich etwas über die interne Repräsentation bestimmter Daten "erforschen" möchte. Ein Beispiel ist Programm 9.1, das die rechnerinterne Darstellung von REAl-Zahlen hexadezimal ausdruckt. Ich komme im Abschnitt 11.1 auf inhaltliche Beobachtungen zurück.

Die Prozedur 'InOut.WriteHex' druckt CARDINAL-Werte hexadezimal aus (s. Programm 2.3, Zeilen 106 bis 111). Für REAL-Werte ist sie nicht zu gebrauchen. Die Idee von Programm 9.1 besteht darin, in einem Datensatz zwei Varianten zu überlagern: Die eine enthält ein REAL-Datenfeld, die andere CARDINAL-Datenfelder. Ich kann einen REAL-Wert hineinschreiben und CARDINAL-Werte herauslesen. Die Datensatzvariable 'Trick' (Zeilen 9 bis 14) leistet das Verlangte.

Es gibt keine gemeinsamen Datenfelder, auch kein Auswahlfeld. Aus der Dokumentation meines MODULA-2-Systems weiss ich, dass eine REAL-Grösse doppelt soviel Speicherplatz benötigt wie eine CARDINAL-Grösse. Deshalb enthält die FALSE-Variante zwei CARDINAL-Datenfelder.

Ich rate sehr von der routinemässigen Benutzung solcher Tricks ab. Bedenken Sie, dass diese Überlagerungen immer von der Maschine und den dort verfügbaren Datenformaten abhängen! Es ist Zufall, wenn ein solches Programm auch auf einem anderen Rechner funktioniert.

Der alleinstehende Doppelpunkt vor dem Typ BOOLEAN in Zeile 10 wurde erst bei der Sprachrevision von 1984 [Wirt84] aufgenommen. Er soll sofort deutlich machen, was dort weggelassen wurde. Bei älteren MODULA-2-Compilern darf der Doppelpunkt nicht geschrieben werden. (Meiner verträgt den Doppelpunkt auch nicht, aber ich habe ihn der Ordnung halber nachträglich eingefügt.)

```
 1 MODULE RealRepraesentation;
 2 (*-------------------------------------------------------------
 3          Hexadezimale Ausgabe einiger REAL-Zahlen
 4 ---------------------------------------------------------------*)
 5 FROM InOut IMPORT WriteHex, WriteInt, WriteLn;
 6 (*---------------------------------------------------------------*)
 7 VAR
 8   i: INTEGER;
 9   Trick: RECORD
10            CASE : BOOLEAN OF
11              TRUE   r: REAL;
12            | FALSE: c1, c2: CARDINAL;
13            END;
14          END;
15 CONST
16   Min = -3; Max = 17; Breite = 5;
17 (*---------------------------------------------------------------*)
18 BEGIN
19   FOR i:=Min TO Max DO
20     WITH Trick DO
21       r := FLOAT(i); WriteInt(i,6);
22       WriteHex(c1,Breite+3); WriteHex(c2,Breite); WriteLn;
23     END;
24   END;
25 END RealRepraesentation.
```

Programm 9.1. RECORD-Typ mit Varianten ohne Auswahlfeld

Die ebenfalls erst mit [Wirt84] eingeführte syntaktische Spitzfindigkeit, dass die "leere" 'variante' erlaubt ist, unterstützt eine bez. "Interpunktion" einheitliche Schreibweise von Programmen.
RECORD-Typen sind eines der meistgebrauchten Ausdrucksmittel von MODULA-2. Die Sortierprogramme im Abschnitt 10.4 arbeiten über ARRAY-Strukturen, deren Komponenten Datensätze sind. Das Pointerkonzept (s. Abschnitt 12) wäre ohne Records nahezu sinnlos.

9.2.2. ARRAY-Typen, offene ARRAY-Parameter und Zeichenketten

ARRAY-Typen sind Datenstrukturen, die für jedes Element ihres "Indextyps" (bzw. für jede Kombination von Werten ihrer Indextypen) genau ein Element ihres "Komponententyps" enthalten. Das heisst, im Gegensatz zu den RECORD-Typen haben alle Komponenten denselben Typ.

```
$    array_typ = ARRAY indextyp {"," indextyp} OF typ .
$    indextyp = q_name | diskreter_typ .
```

Der 'q_name' muss sich auf einen Aufzählungstyp, einen Teilbereichstyp oder einen der Typen CHAR und BOOLEAN beziehen.

Der 'typ' hinter OF ist der Komponententyp. Ich betrachte zunächst nur solche ARRAY-Typen, bei denen genau ein 'indextyp' angegeben ist; man spricht von eindimensionalen Arrays. Es folgen Beispiele:

```
CONST
  KettenLaenge = 32;
TYPE
  Kette = ARRAY [0..KettenLaenge-1] OF CHAR;
  Nummern = [0..9];
  Vektor = ARRAY Nummern OF REAL;
VAR
  H: ARRAY CHAR OF INTEGER;
  text: Kette;  loesung,r: Vektor;
```

Die Werte des Typs 'Kette' haben 32 Komponenten. Jede ist vom Typ CHAR. Vektoren bestehen aus zehn REAL-Komponenten. Die Variable 'H' enthält 128 INTEGER-Komponenten: für jeden Wert des Typs CHAR eine. Die Auswahl einer ARRAY-Komponente erfolgt durch Angabe des jeweiligen Indexwerts (bzw. der Wertekombination) in eckigen Klammern.

```
$    array_selektor = "[" index {"," index} "]"
$    index = ausdruck .
```

Bei eindimensionalen Arrays muss genau ein mit dem entsprechenden 'indextyp' zuweisungskompatibler 'indexausdruck' angegeben werden.

```
H["@"]   text[17]   loesung[i+1]   H[text[KettenLaenge-2]]
```

Die Beispielvariable 'H' kommt in dem als Programm 9.2 abgedruckten

Programmodul 'Haeufigkeiten' vor. 'H[z]' bedeutet dort die Häufigkeit des Auftretens von Zeichen 'z'.

```
 1 MODULE Haeufigkeiten;
 2 (*-------------------------------------------------------------
 3   Ermitteln und Ausschreiben der Haeufigkeiten des Auftretens
 4   von Zeichen im File 'haeuf.dat'
 5 -------------------------------------------------------------*)
 6 FROM InOut IMPORT
 7   OpenInput, CloseInput, Done, Read, Write, WriteInt, WriteLn;
 8 (*-------------------------------------------------------------*)
 9 VAR
10   H: ARRAY CHAR OF INTEGER; (* Haeufigkeiten; Anf.-wert: 0 *)
11   z: CHAR;  n: INTEGER;
12 PROCEDURE Maximum(VAR z: CHAR; VAR max: INTEGER): BOOLEAN;
13   (* Aufsuchen des Maximums im Array 'H' und "Vernichten"
14      des Eintrags.
15      Resultat:   'TRUE'  - Maximale Anzahl ist groesser als 0
16                  'FALSE' - Keine Anzahl ist groesser als 0   *)
17 VAR zz: CHAR;
18 BEGIN
19   z:=0C; max:=H[0C];
20   FOR zz:=1C TO 177C DO
21     IF H[zz]>max THEN z:=zz; max:=H[zz] END;
22   END;
23   IF max<=0 THEN RETURN FALSE END;
24   H[z]:=-1; RETURN TRUE;
25 END Maximum;
26 (*-------------------------------------------------------------*)
27 BEGIN
28   FOR z:=0C TO 177C DO H[z]:=0 END;
29   OpenInput("haeuf.dat");
30   LOOP Read(z); IF NOT Done THEN EXIT END; INC(H[z]) END;
31   CloseInput;
32   WHILE Maximum(z,n) DO Write(z); WriteInt(n,6); WriteLn END;
33 END Haeufigkeiten.
```

Programm 9.2. Ermitteln von Zeichenhäufigkeiten

Das Belegen eines Arrays mit Konstanten kann nur komponentenweise erfolgen, wie das z. B. in Zeile 28 von Programm 9.2 geschieht. Es ist bei MODULA-2 i. allg. nicht möglich, "ARRAY-Konstanten" zu formulieren. Die einzige Ausnahme (Zeichenkettenkonstanten) wird noch in diesem Abschnitt ausführlich behandelt. Analog zu RECORD-Typen können aber Wertzuweisungen zwischen ARRAY-Variablen (bzw. RECORD- oder ARRAY-Komponenten, die selbst Arrays sind) ausgeführt werden. Wenn alle Komponenten der Variablen 'loesung' und 'r' auf Null gesetzt werden sollen, dann kann das mit folgenden Anweisungen geschehen. 'i' sei vom Typ INTEGER.

```
FOR i:=0 TO 9 DO loesung[i]:=0.0 END;
r:=loesung;
```

Die letzte Ergibtanweisung belegt das Array 'r' "auf einen Schlag" mit dem Wert des Arrays 'loesung'.

Als Typangabe für formale Parameter von Prozeduren ist die Schreibweise

```
ARRAY OF typname
```

zugelassen (s. Syntaxregel 'formaler_typ' im Abschn. 5.2). Parameter mit derartigen Typangaben werden als offene Arrayparameter oder dynamische Arrayparameter oder auch kürzer als offene bzw. dynamische Arrays bezeichnet. In einem offenen Array ist offenbar der Indextyp offengelassen worden.

> Auf der Position eines offenen Arrayparameters sind als aktuelle Parameter beliebige eindimensionale Arrays mit dem hinter "ARRAY OF" angegebenen Komponententyp erlaubt.

Innerhalb der Prozedur, in deren Kopf ein dynamisches Array auftritt, erscheint dieser Parameter so, als hätte er den Indextyp

```
CARDINAL[0..Anzahl-1],
```

wobei 'Anzahl' die Anzahl der Elemente des Indextyps im jeweiligen aktuellen Parameter bedeutet. Man benötigt eine innerhalb der betreffenden Prozedur funktionierende Möglichkeit zur Feststellung des von Aufruf zu Aufruf veränderlichen Wertes 'Anzahl-1'. Diesem Zweck dient die Standardfunktion HIGH. Als aktueller Parameter ist der Name eines offenen Arrays anzugeben. Der Funktionsaufruf

```
HIGH(p)
```

liefert die um 1 verringerte Komponentenanzahl des zum offenen Array p gehörenden aktuellen Parameters als CARDINAL-Zahl.

Betrachten wir ein Beispiel! 'MaxAbs' ist eine Funktionsprozedur, die das Maximum der Absolutbeträge der Komponenten von INTEGER-Arrays bestimmt. HIGH tritt in Zeile 5 auf. Die Typtransferfunktion INTEGER(...) ist erforderlich, weil ich CARDINAL-Variablen meide.

```
1 PROCEDURE MaxAbs(VAR a: ARRAY OF INTEGER): INTEGER;
2 VAR  i, max: INTEGER;
3 BEGIN
4   max:=ABS(a[0]);
5   FOR i:=1 TO INTEGER(HIGH(a)) DO
6     IF ABS(a[i])>max THEN max:=ABS(a[i]) END;
7   END;
8   RETURN max;
9 END MaxAbs;
```

Diese Funktion ist variabel einsetzbar. Sie funktioniert nicht nur für genau einen Parametertyp, sondern sie kann alle möglichen eindimensionalen Arrays mit INTEGER-Komponenten bearbeiten. Wenn wir die Variablenvereinbarungen

```
VAR
    Tageseinnahmen: ARRAY [1..31] OF INTEGER;
    Monatswerte: ARRAY Monat OF INTEGER;
    xyz: ARRAY [-7..3] OF INTEGER;
```

voraussetzen, dann sind folgende drei Funktionsaufrufe korrekt:

```
MaxAbs(Tageseinnahmen)   MaxAbs(Monatswerte)   MaxAbs(xyz)
```

Auch auf die im Programm 9.2 vorkommende Arrayvariable 'H' könnte 'MaxAbs' angesetzt werden: `MaxAbs(H)`.

Die jeweiligen von HIGH(a) (s. Zeile 5) gelieferten Werte sind in der Reihenfolge der notierten Aufrufe 30, 11, 10 und 127.

Offene Arrayparameter unterliegen Benutzungsbeschränkungen. Es ist verboten, Wertzuweisungen an oder mit offenen Arrays "als Ganzes" auszuführen. Als aktuelle Parameter dürfen sie nur auf der Position eines passenden offenen Arrays verwendet werden. Drei Regeln:

1. Ein offener Arrayparameter erlaubt nur Bezugnahmen auf seine einzelnen Komponenten.
2. Ein offener Arrayparameter kann als aktueller Parameter für einen offenen Arrayparameter "weitergereicht" werden.
3. Was nicht nach Punkt 1 oder 2 erlaubt ist, ist verboten.

Warum habe ich im Kopf der Prozedur 'MaxAbs' einen Referenz- und keinen Wertparameter notiert? Das ist eine Frage der Effektivität,

die jeder Programmierer bei der Arbeit mit Arrayparametern gründlich bedenken sollte. Die Prozedur 'MaxAbs' würde genauso funktionieren, wenn das VAR in Zeile 1 weggelassen wäre. In den folgenden Überlegungen spielt es keine Rolle, ob offene oder "gewöhnliche" Arrayparameter vorliegen.
Wertparameter sind lokale Objekte der Prozedur, die beim Aufruf mit einem Anfangswert (aktueller Parameter) belegt werden. Für ein solches Array muss also Speicherplatz vorgesehen werden, und ausserdem muss bei jedem Aufruf der gesamte aktuelle Parameter umkopiert werden. Beides organisiert der Compiler automatisch. Aber das Umkopieren kostet Zeit! Und bei grossen Arrays kann auch der Speicherplatzbedarf zum Problem werden. Bei Referenzparametern gibt es diese Probleme nicht. Ich empfehle deshalb,

> ARRAY-wertige Wertparameter nur dann zu verwenden, wenn es wichtig ist, dass eine lokale Kopie entsteht und der aktuelle Parameter "unzerstört" bleibt.

In meiner Programmierpraxis traten solche Situationen sehr selten auf. Beim Programmieren mit Arrays ist ein weiterer Punkt bedenkenswert. Er hat aber i. allg. keine so grosse Wirkung auf die Abarbeitungsgeschwindigkeit wie die Frage der Parameterart. Ich ziehe Arrays vor, deren kleinster Index 0 bzw. das kleinste Element eines Aufzählungstyps ist. Wie wird der Compiler Bezugnahmen auf Arraykomponenten organisieren?
Gewiss steht irgendwo die Anfangsadresse des Arrays. Aus dem gewünschten Index und der unteren Indexgrenze ergibt sich durch Subtraktion, das wievielte Element gesucht ist. Die zur unteren Indexgrenze gehörende Arraykomponente wird als "nulltes" Element angesehen. Nehmen wir an, das n-te Element sei gesucht. n wird mit der Elementlänge des Komponententyps multipliziert, und durch Addition des Ergebnisses zur Anfangsadresse ist man bei der gewünschten Komponente.
Summa summarum: Hinter dem Zugriff auf eine Komponente eines eindimensionalen Arrays verbergen sich eine Subtraktion, eine Multiplikation und eine Addition. Die Subtraktion wird genau dann trivial, wenn die untere Indexgrenze gleich Null ist. Fast jeder Compiler erkennt diesen Spezialfall und lässt die Subtraktion weg.
Wenn wir voraussetzen, dass ein guter Compiler Wege zur besonders schnellen Multiplikation mit festen Zweierpotenzen kennt, dann sind

Arrays, deren Komponententyp einen Speicherplatzbedarf von 2**m Einheiten hat, besonders laufzeitgünstig. Ein vernünftiger Compiler wird natürlich bei Komponentenlänge 1 (vielleicht Typ CHAR) die Multiplikation weglassen.
Versprechen Sie sich keine zu grossen Effekte von Änderungen der unteren Indexgrösse und der Komponentenlänge! Durch solche Massnahmen können zwar gute Programme noch ein wenig besser werden; aber ein langsames Programm bleibt ein langsames Programm. Oft muss über das Lösungsverfahren und die Aufgabe nachgedacht und nachgelesen werden (z. B. in [HoSa81]), um Laufzeitverbesserungen zu erreichen oder herauszufinden, dass es prinzipiell nicht schneller geht.

MODULA-2 bietet keine Standardprozeduren oder speziellen Operatoren für die Arbeit mit Zeichenketten an. Lediglich bei Ergibtanweisungen und Parametervermittlung wurden einschlägige Sonderregelungen in die Sprachdefinition aufgenommen. Sie betreffen Variablen, Komponenten und Wertparameter der Typen

```
TN = ARRAY [0..N-1] OF CHAR
```

sowie Wertparameter des Typs

```
ARRAY OF CHAR.
```

Ich fasse diese Regelungen in vier Punkten zusammen und bringe anschliessend ein Programmierbeispiel.

1. Zeichenkettenkonstanten der Länge L (vgl. Abschn. 4.2) werden als Konstanten des Typs

   ```
   ARRAY [0..L-1] OF CHAR
   ```

   angesehen. Das sind die einzigen in MODULA-2 möglichen Arraykonstanten.
2. Einer Variablen v des Typs TN können auch Zeichenkettenkonstanten zugewiesen werden, deren Länge nicht grösser als N ist. Bei Wertzuweisungen von Zeichenkettenkonstanten einer Länge L mit L<N wird nach dem letzten Zeichen der Konstante ein Steuerzeichen 0C ('nul', s. Tafeln 4.4 und 4.5) als Endekennzeichen angefügt.
3. Bei formalen Werteparametern vom Typ TN sind als aktuelle Parameter auch Zeichenkettenkonstanten zulässig, deren Länge L nicht grösser als N ist. Wenn der aktuelle Parameter kürzer ist (d. h. L<N), dann wird im formalen Parameter nach dem letzten Zeichen der Konstante noch ein 0C hinzugefügt.

4. Wenn eine Zeichenkettenkonstante der Länge L als aktueller Parameter für einen Wertparameter des Typs ARRAY OF CHAR substituiert wird, dann liefert HIGH den Wert L-1. Es wird kein 0C angehängt.

Das Endekennzeichen 0C spielt in den Ausgabeprozeduren für Zeichenketten eine wichtige Rolle. Diese Prozeduren halten sich an die Konvention, dass eine Zeichenkette zeichenweise ausgeschrieben wird, bis entweder

- ihr Ende erreicht ist oder
- ein 0C vorliegt.

Beispielsweise hat die Prozedur 'WriteString' des Bausteins 'InOut' bei mir folgendes Aussehen:

```
PROCEDURE WriteString(s: ARRAY OF CHAR);
VAR i: CARDINAL;
BEGIN
  i:=0:
  WHILE (i<=HIGH(s)) AND (s[i]#0C) DO
    Write(s[i]); INC(i);
  END;
END WriteString;
```

Bemerkung: Die Reihenfolge der Operanden in der WHILE-Bedingung kann nicht vertauscht werden. Bitte überlegen Sie sich, warum das so ist!

Für mich und einige Studenten hat sich das Nichtbeachten aller Konsequenzen aus Regel 4 als böse Falle erwiesen. Was passierte wie? Eine Arrayvariable

```
VAR t: ARRAY [0..15] OF CHAR;
```

wird in einer Prozedur komponentenweise mit den einzelnen Zeichen aus einem offenen Arrayparameter belegt, dem als aktueller Parameter eine Zeichenkettenkonstante zugewiesen wurde. Nehmen wir an, diese Konstante habe die Länge 10. Irgendwann kommt ein Prozeduraufruf

```
InOut.WriteString(t).
```

Auf dem Bildschirm erscheint die erwartete Zeichenkettenkonstante, aber dann kommen noch sechs Zeichen mit "Müll". Wenn im fraglichen Speicherbereich (die letzten sechs Zeichen von 't') zufällig eine Null vorkommt, dann kann der Müll auch kürzer ausfallen oder ganz fehlen.

Die folgende Prozedur 'Kopieren' vermeidet den eben dargestellten heimtückischen Fehler: Sie setzt - wenn nötig - ein 0C ein.

```
 1 PROCEDURE Kopieren(VAR ziel: ARRAY OF CHAR;
 2                    quelle: ARRAY OF CHAR);
 3 VAR i: INTEGER;
 4 BEGIN
 5   i:=0;
 6   LOOP
 7     IF i>INTEGER(HIGH(ziel)) THEN EXIT END;
 8     IF i>INTEGER(HIGH(quelle)) THEN ziel[i]:=0C; EXIT END;
 9     ziel[i]:=quelle[i];
10     IF quelle[i]=0C THEN EXIT END;
11     INC(i);
12   END;
13 END Kopieren;
```

Das Umspeichern der einzelnen Zeichen findet in Zeile 9 statt. Die Schleife hat drei Abbruchbedingungen:

1. Das 'ziel' ist kürzer als die 'quelle' (Zeile 7).
2. Das 'ziel' ist länger als die 'quelle' (Zeile 8).
   In diesem Fall muss auf die aktuelle Position im 'ziel' ein 0C geschrieben werden.
3. In der 'quelle' wird ein 0C gefunden (Zeile 10).
   Dieses Endekennzeichen ist unmittelbar davor in Zeile 9 zum 'ziel' übertragen worden. Also ist alles in Ordnung.

Die Prozedur 'Kopieren' kann auch dann als Vorbild dienen, wenn lexikografische Zeichenkettenvergleiche zu programmieren sind.

Mehrdimensionale ARRAY-Typen werden als abkürzende Schreibweise für verschachtelte eindimensionale Arrays definiert. Mit

```
ARRAY T1, T2, ..., Tn OF T
```

ist der Typ

```
ARRAY T1 OF
  ARRAY T2 OF
    ...
      ARRAY Tn OF T
```

gemeint. Daraus folgt: Bei mehrdimensionalen Arrays gibt es für jede Kombination von Werten der Indextypen ein Arrayelement, und die Anzahl der Elemente eines Arrays mit Indextypen T1, T2, ..., Tn

ist gleich dem Produkt N1*N2*...*Nn, wobei Ni (i = 1, 2, ..., n) die Anzahl der Elemente von Typ Ti angibt. Bezugnahmen auf ein Element eines n-dimensionalen Arrays haben die Form

```
S[I1,I2,...,In],
```

wobei die Indexausdrücke Ii zuweisungskompatibel mit dem Typ Ti sein müssen. Die angegebene Schreibweise ist definiert als Abkürzung für

```
S[I1][I2]...[In].
```

Auch Formen, die zwischen diesen zwei Extremen liegen, wie etwa

```
S[I1,I2][I3]...[In],
```

sind erlaubt. Ich rate von dieser Notationsakrobatik mit mehreren eckigen Klammern ab und empfehle, bei mehrdimensionalen Arrays stets die Schreibweise mit einem einzigen eckigen Klammernpaar zu bevorzugen.

Das bekannteste Beispiel mehrdimensionaler Arrays sind Matrizen.

```
CONST
  Kapazitaet = 10; H = Kapazitaet-1;
TYPE
  Indizes = [0..H];
  Matrix = ARRAY Indizes, Indizes OF REAL;
```

Die folgende Prozedur 'Einheit' belegt den Parameter 'mat' mit einer Einheitsmatrix. Wenn Ihnen der Begriff "Einheitsmatrix" nicht geläufig ist, dann betrachten Sie bitte die Wirkung dieser Prozedur als Begriffsdefinition.

```
 1 PROCEDURE Einheit(VAR mat: Matrix);
 2 VAR
 3   i: INTEGER;
 4 BEGIN
 5   FOR i:=0 TO H DO mat[0,i]:=0.0 END;
 6   FOR i:=1 TO H DO
 7     mat[i]:=mat[0]; mat[i,i]:=1.0;
 8   END;
 9   mat[0,0]:=1.0;
10 END Einheit;
```

Am Anfang wird die erste Zeile der Matrix mit Nullen belegt (FOR-Anweisung auf Zeile 5). Anschliessend speichere ich diese Zeile

jeweils "als Ganzes" auf alle anderen Zeilen um. Die erste Ergibtanweisung in Zeile 7 fusst auf dem Fakt, dass der Typ 'Matrix' definitionsgemäss als

```
ARRAY Indizes OF ARRAY Indizes OF REAL
```

zu verstehen ist. 'm[i]' (und speziell auch 'm[0]') ist also eine erlaubte Schreibweise, die sich auf eine Grösse des Typs

```
ARRAY Indizes OF REAL
```

bezieht. Jeweils nach dem Umspeichern einer Nullzeile wird das Matrixelement 'm[i,i]' auf 1.0 gesetzt. Für 'm[0,0]' muss das am Ende (Zeile 9) nachgeholt werden.
Die Tatsache, dass die Verwendung des Bezeichners einer Matrix mit nur einem Index jeweils eine Zeile der Matrix beschreibt, sollten Sie sich merken.

Betrachten wir als Abschluss des Abschnitts 9.2 über strukturierte Typen noch einmal die Vermischung von RECORD- und ARRAY-Typen. Im Programm 2.2 'Staedte' gibt es auf Zeile 20 die Variable 'tabelle'. Ausgehend von dieser Variablenvereinbarung und der Typvereinbarung 'Eintrag' (s. Zeilen 13 bis 16 von Programm 2.2) sind auf den nächsten Zeilen jeweils links korrekte Bezugnahmen angegeben. Rechts daneben steht der zugehörige Datentyp.

```
tabelle                 ARRAY [0..Kapazitaet-1] OF Eintrag
tabelle[7]              Eintrag
tabelle[7].stadt        ARRAY [0..31] OF CHAR
tabelle[7].leute        INTEGER
tabelle[7].stadt[31]    CHAR
```

Im Bild 2.1 wurden diese Dinge schon einmal mit grafischen Mitteln anschaulich dargestellt.

9.3. SET-Typen

SET- oder Mengentypen werden über diskreten "Basistypen" gebildet.

```
$    set_typ = SET OF basistyp .
$    basistyp = q_name | diskreter_typ .
```
Der 'q_name' muss sich auf einen Aufzählungstyp, einen Teilbereichstyp oder den Typ BOOLEAN beziehen.

Der durch einen Mengentyp beschriebene Wertebereich besteht aus all jenen Mengen, die nur Elemente des Basistyps enthalten. (In der Terminologie der Mathematik würde man sagen: Ein Mengentyp beschreibt die Potenzmenge seines Basistyps.) Randfälle sind die leere Menge, die überhaupt kein Element enthält, und die volle Menge, die aus allen Elementen des Basistyps besteht. Ein Mengentyp hat 2**n verschiedene Werte, wobei n die Elementanzahl des Basistyps ist.
MODULA-2 stellt einen Standard-Mengentyp bereit: BITSET. Er ist definiert als

```
TYPE
  BITSET = SET OF [0..wl-1];
```

wobei 'wl' die Wortlänge des Computers ist. Die üblichen Werte für 'wl' sind 16 (8- und 16-Bit-Rechner) und 32 (32-Bit-Rechner). Der Wert 'wl' stellt auch eine Grössenbeschränkung für alle anderen SET-Typen dar. Wenn T als Basistyp eines Mengentyps erlaubt sein soll, dann muss er die Ungleichungskette

```
0 <= ORD(MIN(T)) <= ORD(MAX(T)) < wl
```

erfüllen. Es folgen Beispiele für Vereinbarungen von SET-Typen:

```
TYPE
  Merkmale = (gross,schwer,bunt);
  Operationen = (loeschen,positionieren,lesen,schreiben);
  Kennung = SET OF Merkmale;
  Kommando = SET OF Operationen;
  MiniZahlen = SET OF [0..9];
```

Mengen werden durch Aufzählen ihrer Elemente in geschweiften Klammern notiert. Wenn aufeinanderfolgende Elemente drin sein sollen, dann gibt es eine abkürzende Intervallschreibweise. Das Hilfssymbol

'menge' trat im Abschnitt 7.1 als 'elementaroperand' innerhalb von Ausdrücken auf.

```
$    menge = [q_name] "{" [element {"," element}] "}"
   Der 'q_name' muss einen SET-Typ bezeichnen.
$    element = ausdruck [".." ausdruck]
```

Der 'q_name' gibt an, zu welchem Mengentyp die Menge gehören soll. Fehlt er, dann ist die Menge vom Typ BITSET. Die Werte der in 'element' auftretenden Ausdrücke müssen aus dem Basistyp der Menge sein. Bei Intervallen darf der Wert des zweiten Ausdrucks nicht kleiner sein als der des ersten.
Beispiele für korrekt gebildete Mengen:

```
    {}                                    {3, 5..8}
    Kennung{}                             Kennung{gross,schwer}
    Kommando{loeschen,lesen,schreiben}    MiniZahlen{3, 5..8}
```

Als erstes habe ich die leere Menge des Typs BITSET aufgeschrieben. Auch die danebenstehende Menge ist vom Typ BITSET. Die letzte Menge enthält zwar die gleichen Elemente, sie hat aber den Typ 'Mini-Zahlen'. Das zweite Beispiel in der linken Spalte beschreibt die leere Menge des Typs 'Kennung'. Versuche zu Mengenkonstruktionen der Art {loeschen,lesen,schreiben} oder {gross,schwer} werden vom Compiler zurückgewiesen. Bei mir lautet die Mitteilung

```
    97: type incompatible constructor element.
```

Bei der Arbeit mit strukturierten Datentypen (RECORD, ARRAY) greift man auf die Komponenten zu; man ermittelt ihren Wert, und man kann ihn verändern. Bei SET-Typen interessiert nur, ob ein Element in der Menge drin ist oder nicht.
Die für Mengen definierten Operatoren und Standardprozeduren testen oder beeinflussen die Mengenzugehörigkeit von Elementen. Mengenoperationen werden durch Operationssymbole beschrieben, die bis auf eine Ausnahme bereits als arithmetische Operatoren bekannt sind.

+	*	Vereinigung und Durchschnitt
-	/	Differenz und symmetrische Differenz
<=	>=	Teilmengenrelation
=	# <>	Test auf Gleichheit bzw. Ungleichheit
IN		Elementrelation

Ein Ausdruck a IN m liefert genau dann das Ergebnis 'TRUE', wenn

der Wert des Ausdrucks a in der durch m beschriebenen Menge enthalten ist. Der Typ des zweiten Operanden von IN muss ein Mengentyp sein. Der erste Operand muss vom Basistyp des zweiten sein.
Bei allen anderen Mengenoperationen müssen beide Operanden von demselben Mengentyp sein. Der Typ des Resultats ist bei +, *, - und / gleich dem Operandentyp. Die Elementrelation IN kann zur exakten Definition der Operationen benutzt werden:

1. Die **Vereinigung** a+b der Mengen a und b besteht aus allen Elementen, die in mindestens einer der beiden Mengen vorkommen.
   x IN (a+b) genau dann, wenn (x IN a) OR (x IN b)
2. Der **Durchschnitt** a*b zweier Mengen a und b besteht aus allen Elementen, die in beiden Mengen auftreten.
   x IN (a*b) genau dann, wenn (x IN a) AND (x IN b)
3. Die **Differenz** a-b der Mengen a und b besteht aus allen Elementen von a, die nicht zu b gehören.
   x IN (a-b) genau dann, wenn (x IN a) AND NOT(x IN b)
4. Die **symmetrische Differenz** a/b der Mengen a und b besteht aus allen Elementen, die in genau einer der zwei Mengen vorkommen.
   x IN (a/b) genau dann, wenn (x IN a) # (x IN b)
   Anders ausgedrückt: a/b = (a+b)-(a*b).
5. Die Menge a ist eine **Teilmenge** von b (a<=b), wenn alle Elemente der Menge a auch zu b gehören.
   a<=b genau dann, wenn für alle x mit x IN a gilt: x IN b
6. Die Menge a ist eine **Obermenge** von b (a>=b), wenn alle Elemente der Menge b auch zu a gehören.
   a>=b genau dann, wenn für alle x mit x IN b gilt: x IN a
   Anders ausgedrückt: a>=b genau dann, wenn b<=a.

Zwei Mengen sind gleich, wenn sie dieselben Elemente enthalten; andernfalls sind sie ungleich. Die Operatoren <= und >= schliessen Mengengleichheit ein. Für den Test, ob a eine "echte" Teilmenge von b ist, muss ein zusammengesetzter Ausdruck benutzt werden:

```
(a<=b) AND (a#b).
```

Sei v eine mengenwertige Variable bzw. eine mengenwertige Komponente eines RECORD- oder ARRAY-Typs. Für die Hinzunahme und die Wegnahme eines einzelnen Elements x gibt es zwei Standardprozeduren.

```
INCL(v,x)          v := v+{x};
EXCL(v,x)          v := v-{x};
```

x muss zuweisungskompatibel zum Basistyp der Menge v sein.

Für ein paar Beispiele seien Variablen

```
VAR
  elefant: Kennung;
  r,s,t: BITSET;
  i: CARDINAL;
```

und Wertzuweisungen

```
elefant := Kennung{gross..bunt}; EXCL(elefant,bunt);
r := {1, 4..9, 13}; s := {2..7, 11..14}; t := {};
i := 3;
```

gegeben. Folgende Vergleichsausdrücke liefern den Wert 'TRUE':

```
gross IN elefant                  2*i IN r
r+s = {1..9,11..14}               r*s = {4..7,13}
r-s = {1,8,9}                     s-r = {2,3,11,12,14}
r/s = {1..3,8,9,11,12,14}         t <= r
r*{1,i..2*i-1} = {1,4,5}          t+{2*i..3*i+2} = {6..11}
```

Die innerhalb von Konstantenausdrücken auftretenden Konstruktionen 'konst_menge' unterscheiden sich von den Mengen dadurch, dass in den Elementen nur Konstantenausdrücke vorkommen dürfen.

```
$    konst_menge =
$           [q_name] "{" konst_element {"," konst_element} "}"
   Der 'q_name' muss einen SET-Typ bezeichnen.
$    konst_element = konstantenausdruck [".." konstantenausdruck]
```

Alle Regeln bez. der Werte der Konstantenausdrücke sind analog zu 'menge'.

Die Unterscheidung von 'menge' und 'konst_menge' gibt es erst in der neueren Sprachversion [Wirt84]. Vorher existierte nur das, was jetzt 'konst_menge' heisst. Die in den Beispielen enthaltenen Mengenkonstruktionen {1,i..2*i-1} und {2*i..3*i+2} werden deshalb von älteren Compilern nicht akzeptiert.

Die Standardprozeduren INCL und EXCL sind i. allg. schneller als die gleichbedeutenden Ergibtanweisungen, weil moderne Rechner die verlangte Operation mit spezialisierten Maschinenbefehlen ausführen können. Die gegenüber den meisten PASCAL-Implementationen arge Verkürzung der Mengentypen zielt generell auf die Verwendbarkeit einzelner Maschinenbefehle für Mengenoperationen ab: Anpassung an die Verarbeitungsbreite der Rechner.

Mengen werden intern als Bitmuster in einem Maschinenwort dargestellt. Jedem Element des Basistyps entspricht ein Bit. Wenn ein Element in einer Menge enthalten ist, so hat "sein" Bit den Wert 1, sonst hat es den Wert 0. Wenn die Werte 1 und 0 als 'TRUE' und 'FALSE' interpretiert werden, dann kann man die Mengenoperationen

Vereinigung	als bitweises OR
Durchschnitt	als bitweises AND
symmetrische Differenz	als bitweises XOR

verstehen. XOR bedeutet "ausschliessendes (exklusives) OR". Ob die Bits einer Menge den Elementen des Basistyps von links nach rechts oder von rechts nach links zugeordnet sind, das ist implementationsabhängig und für den Programmierer uninteressant, solange er sich nur innerhalb der Sprache MODULA-2 bewegt. Wenn man allerdings mit Mengen auf bestimmte, schon in anderen Zusammenhängen und Sprachen fixierte Bitmuster Bezug nehmen will, dann muss man die Reihenfolge kennen. Eine am Programm 9.1 orientierte "Untersuchung" hilft.

Mengentypen sind vorteilhaft einsetzbar, wenn Bitmusteroperationen ausgeführt werden sollen (z. B. Schwarz/Weiss-Grafik) oder wenn Zusammenstellungen von "Kennzeichen" zu bearbeiten sind (s. die Typen 'Kennung' und 'Kommando'). Einem Riesenluftballon würde man ggf. die Menge 'Kennung{gross,bunt}' zuordnen. Eine Prozedur zur Abarbeitung von aus mehreren Operationen zusammengesetzten Kommandos könnte folgenden prinzipiellen Aufbau haben:

```
PROCEDURE Ausfuehren(k: Kommando);
VAR o: Operationen;
BEGIN
  FOR o:=loeschen TO schreiben DO
    IF o IN k THEN ... END;
  END;
END Ausfuehren;
```

Manchmal lassen sich auch spezielle Bedingungen, denen ein Wert genügen soll, elegant mit Mengen formulieren. An Stelle von

```
(i=3) OR (i>=6) AND (i<11) OR (i=13) OR (i=15)
```

könnte man schreiben

```
i IN {3,6..10,13,15}.
```

Leider ist das Programmverhalten für Werte von 'i', die grösser als die erlaubte Mengenobergrenze sind, nicht generell festgelegt.

9.4. Prozedurtypen

Ein Prozedurtyp beschreibt Prozeduren mit "gleichartigen" Prozedurköpfen. Gleichartig bedeutet

1. gleiche Parameteranzahl
2. gleiche Reihenfolge bez. Parameterart und -typ
3. gleichen Resultattyp (ggf. gar keinen).

Das heisst, ein Prozedurtyp ist durch die im Abschnitt 5.3 eingeführte abstrakte Signatur charakterisiert:

```
$    prozedur_typ = PROCEDURE abstrakte_signatur .
```

Der Standardtyp PROC (s. Abschn. 6.6) ist definiert als

```
TYPE
  PROC = PROCEDURE;
```

Zur Speicherung "prozeduraler Symptome" wurde im Typ 'Symptom' (s. Abschn. 9.2.1) ein Datenfeld mit dem Typ 'TestProzedur' eingeführt. Die zugehörige Typvereinbarung lautet:

```
TYPE
  TestProzedur = PROCEDURE(): BOOLEAN;
```

Das heisst, bei dem erwähnten Expertensystem zur Diagnose technischer Geräte müssen die On-line-Messwerterfassungen in die Form parameterloser Funktionsprozeduren gebracht werden, die BOOLEAN-Resultate ("zufrieden mit den Messwerten"/"unzufrieden") liefern. Weitere Beispiele:

```
TYPE
  ReadProc = PROCEDURE(VAR CHAR);
  WriteProc = PROCEDURE(CHAR);
  MathFunktion = PROCEDURE(REAL): REAL;
  StringVergleich =
            PROCEDURE(ARRAY OF CHAR, ARRAY OF CHAR): BOOLEAN;
```

Für Werte aus Prozedurtypen sind keine Operationen erklärt. Man kann mit ihnen genau zwei Dinge tun:

- an prozedurwertige Variablen oder RECORD- bzw. ARRAY-Komponenten zuweisen und
- abarbeiten (Funktions- bzw. Prozeduraufruf).

Bei Ergibtanweisungen mit Prozeduren und bei prozedurwertigen Wert-

parametern gibt es eine wichtige Beschränkung. Auf der rechten Seite bzw. als aktueller Parameter dürfen

1. Prozedurvariablen bzw. prozedurwertige RECORD- und ARRAY-Komponenten,
2. von anderen Bausteinen importierte Prozeduren (s. Abschn. 10) und
3. solche Prozeduren, die lokal zum Modulblock vereinbart worden sind,

auftreten. Diese Regel schliesst gerade jene Prozeduren aus, deren Vereinbarungen sich innerhalb anderer Prozeduren befinden.

Programm 6.1 'Enden' im Abschnitt 6.6 ist ein Beispiel für den gewinnbringenden Einsatz von Prozedurvariablen. Zur Vertiefung des dort vermittelten Eindrucks möchte ich jetzt zeigen, wie die Nutzbarkeit von Datenkonvertierungsprozeduren durch Rückgriff auf geeignete Prozedurvariablen wesentlich vergrössert werden kann. Ich kopiere die Prozedur 'LiesInteger' (Programm 8.1), gebe ihr zusätzliche Parameter 'READ' und 'WRITE' vom Typ 'ReadProc' bzw. 'WriteProc' und benutze diese Parameter an Stelle von 'InOut.Read' und 'InOut.Write'.

```
 1 PROCEDURE LiesInteger(READ: ReadProc;
 2                       WRITE: WriteProc;
 3                       VAR wert: INTEGER);
 4 VAR
 5   z: CHAR;  negativ: BOOLEAN;
 6 BEGIN
 7   wert := 0;
 8   LOOP READ(z); IF z>" " THEN EXIT END END;
 9   IF (z="+")OR(z="-") THEN
10     negativ := z="-"; WRITE(z); READ(z);
11   ELSE negativ := FALSE END;
12   LOOP IF (z<"0") OR (z>"9") THEN EXIT END;
13     wert := wert*10 + INTEGER(ORD(z)-ORD("0"));
14     WRITE(z); READ(z);
15   END;
16   IF negativ THEN wert := -wert END;
17 END LiesInteger;
```

Was habe ich davon? Ich kann auf elegante Weise die Quelle der einzulesenden Zeichen und den "Platz" für das Protokoll ändern.

Wenn bei einem Aufruf die 'InOut'-Prozeduren zum Lesen und Schreiben einzelner Zeichen als aktuelle Parameter eingesetzt werden, dann läuft alles wie vorher.

```
LiesInteger(InOut.Read, InOut.Write, n);
```

Nun seien folgende Variablen- und Prozedurvereinbarungen gegeben:

```
VAR
  kette: ARRAY [0..31] OF CHAR;
  kPosition: INTEGER;
PROCEDURE NaechstesZeichen(VAR z: CHAR);
  BEGIN
    IF kPosition>31 THEN z:=0C; RETURN END;
    z:=kette[kPosition];
    IF z#0C THEN INC(kPosition) END;
  END NaechstesZeichen;
PROCEDURE Senke(z: CHAR);
  BEGIN  END Senke;
```

Wenn ich nach dem Ausführen der "Initialisierungsanweisungen"

```
kPosition:=0;   kette:='   123   -777';
```

die Prozeduraufrufe

```
LiesInteger(NaechstesZeichen, Senke, n);   und
LiesInteger(NaechstesZeichen, InOut.Write, n);
```

abarbeite, dann werden Zeichen aus der Variablen 'kette' konvertiert. Beim ersten Aufruf fällt das Protokoll in die Senke. Beim zweiten sehe ich an der übliche Stelle die '-777'.

Ich empfehle,

beim Programmentwurf die Benutzung geeigneter Prozedurvariablen und prozedurwertiger RECORD- bzw. ARRAY-Komponenten in Erwägung zu ziehen.

Sammeln Sie Erfahrungen damit; es lohnt sich!

Die einzige noch ausstehende Familie von MODULA-2-Datentypen, die Pointertypen, sind Gegenstand von Abschnitt 12.

9.5. Typgleichheit, Kompatibilität und Zuweisungskompatibilität

Im Abschnitt 6 (Standardtypen) wurde für zweistellige Operationen Typgleichheit der Operanden gefordert. Teilbereichstypen erzwingen eine Lockerung dieser Bedingung. Abschnitt 7.3 enthält eine nur vorläufige Definition der Zuweisungskompatibilität.
Ich definiere die Begriffe "Typgleichheit", "Kompatibilität von Ausdrücken" und "Zuweisungskompatibilität eines Objekts und eines Ausdrucks". Anschliessend ist zusammengestellt, an welchen Stellen der Sprache MODULA-2 die jeweilige Verträglichkeitsbedingung eine Rolle spielt.
In diesem Abschnitt benutze ich das Wort "Objekt" als Sammelbegriff für Variablen, formale Parameter, ARRAY- und RECORD-Komponenten sowie Konstanten.

Typgleichheit
Zwei Objekte A und B der Typen TA und TB sind vom gleichen Typ, wenn eine der folgenden fünf Aussagen zutrifft:

1. A und B sind Variablen oder RECORD-Komponenten oder formale Parameter, die bei der Vereinbarung in derselben Bezeichnerliste stehen.
2. A und B sind Komponenten von bei der Vereinbarung in derselben Bezeichnerliste stehenden Arrays.
3. A und B sind Konstanten, die in demselben Aufzählungstyp eingeführt werden.
4. TA und TB werden durch denselben Typnamen bezeichnet. Wenn A und/oder B Konstanten sind, die zu mehreren Typen gehören, dann trifft diese Aussage auf einen der möglichen Typen zu.
5. TA und TB sind Typnamen, und der eine wird in einer Typvereinbarung oder über mehrere Typvereinbarungen als (Alias-)Name für den anderen eingeführt.

Man sagt auch, TA und TB seien gleich.
Beispiele und Erläuterungen:

```
VAR
  x,y: [1..31];  z: [1..31];  i: INTEGER;  c: CARDINAL;
  f,g: ARRAY CHAR OF [1..12];  h: ARRAY CHAR OF [1..12];
```

Nach Regel 1 sind 'x' und 'y' typgleich; aber 'x' und 'z' sowie 'y'

und 'z' sind nicht typgleich. Nach Regel 2 sind die Komponenten von 'f' und 'g' typgleich, aber nicht die von 'f' und 'h' bzw. die von 'g' und 'h'. Die Konstante 123 ist wegen des zweiten Teils von Regel 4 sowohl zu 'i' als auch zu 'c' typgleich.
Bezüglich Regel 5 sind folgende zwei Fälle möglich:

```
TYPE          bzw.   TYPE
  TA = ...;          TA = ...;
  TB = TA;           T1 = TA; T2 = T1; ...; TB = Tn;
```

Kompatibilität
Zwei Ausdrücke mit den Typen T1 und T2 heissen kompatibel, wenn eine der folgenden drei Aussagen erfüllt ist:

1. T1 und T2 sind gleich.
2. T1 ist ein Teilbereichstyp von T2 oder umgekehrt.
3. T1 und T2 sind Teilbereichstypen, deren Basistypen gleich sind.

Man sagt auch, T1 und T2 seien kompatibel.
Beispiele:

```
VAR
  m: INTEGER; n,o: [-100..100];
  u: "a".."z"; v: "A".."Z";
```

'm', 'n' und 'o' sowie 'u' und 'v' sind kompatibel.

Zuweisungskompatibilität
Ein Objekt vom Typ T1, das keine Konstante ist, und ein Ausdruck vom Typ T2 heissen **zuweisungskompatibel**, wenn eine der folgenden fünf Aussagen wahr ist:

1. T1 und T2 sind kompatibel.
2. Einer der beiden Typen ist kompatibel zu INTEGER, und der andere ist kompatibel zu CARDINAL, oder einer ist kompatibel zu LONGINT und der andere zu LONGCARD.
3. T1 ist der Typ ARRAY [0..N-1] OF CHAR, und der Ausdruck ist eine Zeichenkettenkonstante, für deren Länge L gilt: L<=N.
4. T1 ist der Parametertyp ARRAY OF CHAR, und der Ausdruck ist eine Zeichenkettenkonstante.
5. T1 und T2 sind Prozedurtypen mit gleicher Parameteranzahl und gleicher Reihenfolge bez. Typ und Art der Parameter sowie gleichem (ggf. gar keinem) Resultattyp.

Beispiele:

```
CONST
  s1 = "123456789"; s2 = "abcdef";
VAR
  c: CARDINAL;  i: [-100..255];
  s: ARRAY [0..7] OF CHAR;
```

Nach Regel 2 ist 'i' zuweisungskompatibel mit 'c' und umgekehrt. 's2' ist zuweisungskompatibel zur Variablen 's', aber 's1' nicht.

Wo müssen diese drei Verträglichkeitsbedingungen beachtet werden?

Typgleichheit
- aktueller Parameter bei Referenzparametern
- untere und obere Grenze bei Teilbereichstypen

Kompatibilität
- Operanden zweistelliger Operationen in Ausdrücken und Konstantenausdrücken
- Auswahlausdruck und Werte in den Fallkennzeichen bei CASE-Anweisungen
- Laufvariable und Ausdrücke für Anfangs- sowie Endwert bei FOR-Anweisungen
- Typ des Auswahlfeldes und Werte in den Fallkennzeichen bei Variantenteilen in RECORD-Typen
- Elementausdrücke in Mengen und konstanten Mengen

Zuweisungskompatibilität
- aktueller Parameter bei Wertparametern
- linke und rechte Seite bei Wertzuweisungen
- Resultatausdruck bei Funktionsprozeduren
- Indexausdruck und Indextyp bei Bezugnahmen auf Arraykomponenten

Ich bin nicht der Auffassung, dass ein Programmierer diese Liste auswendig kennen muss (auch kein Student in der Prüfung). Aber er muss wissen, dass es diese drei Stufen von Verträglichkeitsbedingungen gibt, um bei Fehlermeldungen seines Compilers in der richtigen Richtung nach Korrekturmöglichkeiten suchen zu können.

10. Das Bausteinkonzept von MODULA-2

Für mich liegt der eigentliche Witz der Sprache MODULA-2 - das ist ihr softwaretechnologischer Wert - im Bausteinkonzept. Zuerst erläutere ich die entsprechenden Sprachelemente. Danach diskutiere ich den Datenfluss bei der Übersetzung von MODULA-2-Programmen. Zum Schluss folgt ein Beispiel, das nicht nur das Bausteinkonzept demonstriert, sondern eine eigenständige Bedeutung hat: Sortieren in Arrays.

10.1. Definitions- und Implementationsmodule

Neben dem bereits im Abschnitt 5.1 eingeführten Programmodul kennt MODULA-2 genau zwei weitere Übersetzungseinheiten: Definitions- und Implementationsmodule.

```
$    uebersetzungseinheit = programmodul
$                         | definitionsmodul
$                         | implementationsmodul .
$    definitionsmodul = DEFINITION MODULE bezeichner ";"
?                       EXPORT QUALIFIED bezeichnerliste ";"
$                       {import ";"} {definition}
$                       END bezeichner "."
   Hinter MODULE und nach dem END muss derselbe 'bezeichner'
   notiert werden (Bausteinname).
$    implementationsmodul = IMPLEMENTATION programmodul .
```

Der syntaktische Rahmen von Definitions- und Implementationsmodulen ist sofort aus den Syntaxregeln ersichtlich. Für verbale Erläuterungen schlage man in den Abschnitten 2.2 und 2.5 nach.
Zu jedem Baustein gehören genau ein Definitionsmodul und mindestens ein Implementationsmodul. Die Verbindung zwischen Definitions- und Implementationsmodul stellt der Compiler über den jeweils hinter MODULE und vor dem abschliessenden Punkt anzugebenden Bausteinnamen her.

Definitionsmodule beschreiben die Schnittstelle eines Bausteins mit seiner "Umgebung". Der Baustein stellt alle in der 'bezeichnerliste' hinter EXPORT QUALIFIED aufgeführten Bezeichner zur Verfügung. Man sagt: Er exportiert sie. Für jeden dieser Bezeichner muss es genau einen Eintrag in einer 'definition' geben. Neben Konstanten-, Variablen- und Typvereinbarungen sind auch "verdeckte Typen" und Prozedurköpfe möglich.

```
$    definition = CONST {konstantenvereinbarung ";"}
$               | VAR {variablenvereinbarung ";"}
$               | TYPE { (typvereinbarung | bezeichner) ";"}
$               | prozedurkopf ";"
```

Verdeckte Typen haben eine 'definition' der Form

```
    TYPE bezeichner ";",
```

die nur festlegt, dass der 'bezeichner' ein Typbezeichner ist. Welcher Typ sich wirklich dahinter verbirgt, das muss im Implementationsmodul stehen.
Alle in den Definitionen auftretenden Konstanten-, Variablen- und Typvereinbarungen sind auch im zugehörigen Implementationsmodul gültig. Sie dürfen dort nicht wiederholt werden.
Die verdeckten Typen und die Prozeduren müssen im Implementationsmodul vollständig vereinbart werden. Bei Prozeduren müssen der Prozedurname, die Parameteranzahl, die Parameterreihenfolge bez. Typ und Art sowie der Resultattyp (ggf. gar keiner) in Implementations- und Definitionsmodul übereinstimmen. Die Parameternamen dürfen also geändert werden; aber man sollte es zwecks Vermeidung von Irrtümern nicht tun.
Bemerkungen:

1. Wenn ein Definitionsmodul einen Aufzählungstyp oder einen RECORD-Typ exportiert, dann werden die Namen der Konstanten des Aufzählungstyps bzw. die Datenfeldnamen des RECORD-Typs automatisch mit exportiert.
2. Die von einem Baustein exportierten Variablen, Konstanten-, Typ- und Prozedurbezeichner müssen alle voneinander verschieden sein. Sie sind auch im zugehörigen Implementationsmodul gültig und gehören zu dessen lokalen Bezeichnern.
3. Ein Definitionsmodul kann Bezeichner importieren. Diese Be-

zeichner muss der Implementationsmodul ebenfalls importieren.

4. Der Export importierter Bezeicher ist verboten.
5. Für die Behandlung von Importen muss der Compiler auf das Resultat der Übersetzung des exportierenden Definitionsmoduls zurückgreifen (vgl. Abschn. 10.2). Aus diesem Grunde sind gegenseitige Importe (A importiert von B und B von A) sowie zyklische Importe (A importiert von B, B von C und C von A) bei Definitionsmodulen nicht möglich.

Mit der Sprachrevision [Wirt84] wurde die EXPORT-QUALIFIED-Klausel in Definitionsmodulen abgeschafft. Das heisst, die mit einem Fragezeichen gekennzeichnete Syntaxzeile wäre zu streichen. Es gilt die Regel: Alle im Definitionsmodul eingeführten Bezeichner werden exportiert (vgl. Bemerkung 4 zu Programm 2.1).

Neuere Compiler verhalten sich zu "alten" Programmen mit EXPORT-QUALIFIED-Klausel entweder liberal (die Klausel wird einfach überlesen und nicht ausgewertet) oder verbissen (Syntaxfehler, z. B. [Robo87b]). Da viele Nutzer noch mit "alten" Systemen arbeiten (z. B. [Logi85]), habe ich die "alte" Syntax angegeben und mit "?" gekennzeichnet. (Das ist die einzige Stelle, wo in diesem Buch von der Sprachfassung [Wirt84] abgewichen wird.)

Bei verdeckten Typen (engl.: hidden types) spricht man auch von verdecktem oder nichttransparentem Typexport. Welche Typen im Implementationsmodul für einen verdeckten Typ angegeben werden können, das hängt vom jeweiligen MODULA-2-System ab. Die Sprachdefinition legt lediglich fest, dass POINTER-Typen (s. Abschn. 12) immer möglich sein müssen. Die Compiler gestatten i. allg. auch INTEGER, CARDINAL und Teilbereiche davon. Manchmal wird formuliert, dass alle Typen zulässig seien, deren Speicherplatzbedarf gleich dem der POINTER-Typen ist. Sehen Sie in Ihrer Nutzerdokumentation nach!

Wenn man einen verdeckten Typ importiert, dann kann man natürlich auch Variablen, ARRAY- bzw. RECORD-Komponenten, formale Parameter und Funktionsprozeduren von diesem Typ vereinbaren. Die Menge der mit solchen Objekten ausführbaren Operationen ist sehr beschränkt:

Sie können auf Gleichheit bzw. Ungleichheit getestet werden. Man kann Wertzuweisungen ausführen und sie als aktuelle Parameter benutzen.

Alle anderen Aktionen mit Objekten, deren Typ verdeckt exportiert

wurde, beispielsweise der Zugriff auf Komponenten, müssen als Prozeduren gemeinsam mit dem Typ exportiert und im Implementationsmodul realisiert werden.
Das ist kein Nachteil, sondern so gewollt. Auf diese Weise kann erreicht werden, dass der Benutzer eines Bausteins keine Chance zu "regelwidrigen" Manipulationen an den vom Baustein intern gehaltenen Daten hat. Er ist gezwungen, nur die vom Baustein verfügbar gemachten Operationen (Prozeduren) zu verwenden. Die einschlägigen softwaretechnologischen Bezugspunkte sind "abstrakter Datentyp" und "Geheimnisprinzip" (vgl. z. B. [Both78], [LiTr87] und die dort angegebene Literatur).

Syntaktisch ist ein Implementationsmodul ein Programmodul mit vorangestelltem Schlüsselwort IMPLEMENTATION. Es wurde bereits festgestellt, dass der Implementationsmodul die Realisierung der Prozeduren und verdeckten Typen des Definitionsmoduls enthalten muss. Welchen Zweck hat jedoch die nach den Syntaxregeln 'programmodul' und 'block' (s. Abschnitt 5.1) vorgesehene 'anweisungsfolge'? Wird sie jemals abgearbeitet? Wenn ja, wann?
Die Anweisungsfolge im Implementationsteil eines Bausteins ist als Folge von Aktionen zur Initialisierung des Bausteins anzusehen. Diese Anweisungen werden abgearbeitet, bevor die Arbeit des Programmoduls beginnt. Bisweilen ist es wichtig, die genaue Initialisierungsreihenfolge zu kennen.
Gegeben sei ein Implementations- oder Programmodul M, der von den Bausteinen M1, M2, Mn importiert, wobei die Importe in dieser Reihenfolge im Programmtext von M stehen. Die Initialisierung von M erfolgt in n+1 Schritten:

> Zuerst werden die Bausteine M1 bis Mn in dieser Reihenfolge initialisiert, dann wird die (ggf. leere) Anweisungsfolge von M abgearbeitet.

Wenn die Bausteine Mi (i=1,...,n) ihrerseits Importe enthalten, dann wird nach demselben Schema verfahren. Die Initialisierung eines Bausteins, der nichts importiert, besteht nur aus der Abarbeitung der Anweisungsfolge (falls vorhanden). Jeder Baustein "merkt" sich, ob seine Initialisierung bereits angefangen hat, und kann auf Grund dieser Information Mehrfachinitialisierungen vermeiden (bedingte Anweisung).

Die Abarbeitung eines MODULA-2-Programms besteht in der "Initialisierung" des Programmoduls.
Betrachten wir ein (hoffentlich) klärendes Beispiel:

```
IMPLEMENTATION MODULE a;          IMPLEMENTATION MODULE b;
  IMPORT b,c;                       IMPORT a,d,c;
  ...                               ...
END a.                            END b.
IMPLEMENTATION MODULE c;          IMPLEMENTATION MODULE d;
  IMPORT d;                         ...
  ...                             END d.
END c.

                    MODULE p;
                      IMPORT a,d;
                      ...
                    END p.
```

Die Abarbeitung des Programmoduls 'p' setzt sich aus den Abarbeitungen der fünf beteiligten Module zusammen. Die "Initialisierung" von 'p' beginnt mit der Initialisierung von 'a'. Die beginnt mit der von 'b'. Und die Initialisierung von 'b' beginnt mit der Initialisierung von 'a', aber die läuft schon, also kommt gleich die Initialisierung von 'd' usw. Letztlich ergibt sich die Abarbeitungsreihenfolge d, c, b, a, p.

10.2. Datenfluss bei der Übersetzung von MODULA-2-Programmen

Sehr grob betrachtet ist ein MODULA-2-Compiler ein Programm, das Files einliest und Files ausgibt.

- Das wesentliche Ergebnis der Übersetzung eines Definitionsmoduls ist ein Symboltabellenfile.
- Bei der Übersetzung von Programm- und Implementationsmodulen entstehen zwei vom MODULA-2-System später noch benötigte Files: ein Objektfile und ein Referenzfile.
- Der Compiler verarbeitet Quelltextfiles. Falls Importe vorkommen, liest er auch Symboltabellenfiles ein.

Darüber hinaus kann ein Compiler noch ein Protokollfile erzeugen. Es ist für die folgenden inhaltlichen Überlegungen uninteressant und wird deshalb erst bei den "Namenskonventionen" wieder erwähnt.

Ein Symboltabellenfile enthält alle nach dem Übersetzen eines Definitionsmoduls bekannten Angaben über die exportierten Bezeichner.
Das Objektfile enthält die aus dem Quelltext hervorgegangenen Maschinenbefehle in einer für den Linker (vgl. Abschnitt 3) "verständlichen" Form. Der Linker erzeugt die abarbeitungsfähige Form eines Programms durch Zusammenfügen des aus dem Programmodul entstandenen Objektfiles mit den Objektfiles aller über Importe einbezogenen Bausteine.
Ein Referenzfile enthält Angaben über die Zuordnung von im Programmtext verwendeten Bezeichnern zu Speicheradressen. Viele MODULA-2-Systeme bieten einen mehr oder weniger komfortablen "Debugger" zur Unterstützung des Programmtests an. Der Debugger greift auf die im Referenzfile gespeicherte Information zurück und kann ermitteln, welchen Wert derzeit das CHAR-wertige Datenfeld 'x.y' hat. Bezüglich genauer Angaben zu den Fähigkeiten des Debuggers verweise ich auf die jeweilige Nutzerdokumentation.

Für jede der drei Übersetzungseinheiten ist der Datenfluss beim Compilieren ein klein wenig anders. Ich unterscheide vier Fälle und notiere sie einheitlich in Tabellenform. Links stehen jeweils Eingabefiles, rechts Ausgabefiles.

1. Definitionsmodul B ohne Import

Quelltext	Symboltabelle B
	Protokoll

2. Programmodul P mit Importen von B1, B2, ..., Bn

Quelltext	Objektfile P
Symboltabelle B1	Referenzfile P
Symboltabelle B2	Protokoll
Symboltabelle Bn	

3. Implementationsmodul C mit Importen von D1, D2, ..., Dm

Quelltext	Objektfile C
Symboltabelle C	Referenzfile C
Symboltabelle D1	Protokoll
Symboltabelle D2	
Symboltabelle Dm	

4. Definitionsmodul E mit Importen von F1, F2, ..., Fk

```
Quelltext                Symboltabelle E
Symboltabelle F1         Protokoll
Symboltabelle F2

Symboltabelle Fk
```

Wir sagen, eine Übersetzungseinheit A sei von einem Baustein B bzw. vom Definitionsmodul B abhängig, wenn

1. A ein Implementationsmodul zu B ist oder wenn
2. A von B importiert.

An dem dargestellten Datenfluss sind sofort Konsequenzen für die zeitliche Reihenfolge von Übersetzungen bzw. die Gültigkeit "alter" Übersetzungsresultate ablesbar:

1. Alle von einem Definitionsmodul X abhängigen Übersetzungseinheiten können erst nach diesem Definitionsmodul übersetzt werden.
2. Bei der Neuübersetzung eines Definitionsmoduls X werden alle vorher entstandenen Übersetzungsresultate, die von X abhängen, ungültig.

Jedes MODULA-2-Programmiersystem muss deshalb über Mittel zur "Versionskontrolle" verfügen. Betrachten wir als Beispiel einen Programmodul 'A', der von einem Baustein 'B' importiert:

```
MODULE A;
  IMPORT B;
  ...
END A.
```

```
DEFINITION MODULE B;          IMPLEMENTATION MODULE B;
  ...                           ...
END B;                        END B.
```

Gemäss Regel 1 muss zuerst der Definitionsmodul 'B' übersetzt werden und danach in beliebiger Reihenfolge der Programmodul 'A' und der Implementationsmodul 'B'. Wenn mir nun eine Änderung am Baustein 'B' einfällt, dann werde ich den Definitionsmodul und danach auch den Implementationsmodul neu übersetzen. Aber das reicht nicht! Die Resultate der Übersetzung des Programmoduls 'A' sind durch die Neuübersetzung des Definitionsmoduls 'B' ungültig geworden. Die Versionskontrolle des Linkers (vgl. Abschn. 3) bemerkt, dass das Objektfile von 'A' nicht zum Objektfile von 'B'

passt. Man übersetze 'A' neu, und schon ist alles in Ordnung.
MODULA-2-Systeme, die eine private Versionskontrolle haben, arbeiten mit einem "Zeit- und Datumsstempel", der beim Übersetzen von Definitionsmodulen in die Symboltabelle eingetragen wird. Darüber hinaus enthält jedes Objektfile und jede Symboltabelle eine Liste mit den Stempeln aller Definitionsmodule, von denen es abhängig ist. Einfacher wird die Sache, wenn man

1. in einem Betriebssystem arbeitet, das für jedes File den Zeitpunkt des letzten Schreibzugriffs im Fileverzeichnis führt, und wenn
2. aus den Programmodul- und Bausteinnamen in fixierter Weise eindeutig auf die Namen der zugehörigen Quelltext-, Symboltabellen- und Objektfiles geschlossen werden kann.

Unter diesen Voraussetzungen kann ein Dienstprogramm ausgehend vom Namen eines Programmoduls alle Abhängigkeiten ermitteln: Man beginne mit einer Inspektion der Importe im Quelltext des Programmoduls und setze diese Inspektion in den Quelltexten der Definitions- und Implementationsmodule jener Bausteine, von denen importiert wird, fort usw. An Hand der so hergestellten Liste kann durch Nachsehen im Filesystem überprüft werden, ob die fraglichen Symboltabellen- und Objektfiles existieren und ob sie zeitlich verträglich sind. Für nicht vorhandene bzw. ungültige Files wird ein Übersetzungsauftrag erzeugt. Abschliessend fügt das Dienstprogramm an die Folge der Übersetzungsaufträge noch einen Auftrag für den Linker an und bringt alles in die Form eines Kommandofiles. Wenn der Nutzer dieses Kommandofile abarbeitet, so hat er am Schluss eine neue abarbeitungsfähige Version seines Programms.
Wenn man ganz am Anfang stand, dann wird alles übersetzt. Ansonsten enthält das entstandene Kommandofile genau die nach der Änderung irgendwelcher Quelltexte notwendigen Compileraufrufe.
Wenn Ihr MODULA-2-System ein solches Werkzeug anbietet, dann sollten Sie es immer benutzen und fast alles, was hier über Versionskontrollen und Übersetzungsreihenfolgen gesagt wurde, wieder vergessen. Der Name derartiger Dienstprogramme ist mit Bezug auf ein bekanntes UNIX-Werkzeug i. allg. 'make' bzw. irgendeine Ableitung davon (z. B. 'm2make' [Logi86]).

10.3. Konventionen für Filenamen

Bei der Arbeit mit MODULA-2 entstehen Datenverwaltungsprobleme: Man hat drei Sorten von Quelltexten sowie Symboltabellen, Objektprogramme, Referenzen, Protokolle und abarbeitungsfähige Programme. MODULA-2 verzichtet auf ein privates Datenverwaltungssystem und nutzt für diese Zwecke das Filesystem des jeweiligen Betriebssystems. Damit das funktioniert, müssen Nutzer und MODULA-2-System gemeinsame feste Regeln für die Ableitung von Filenamen aus Modulnamen einhalten.
Nicht nur das am Schluss des vorigen Abschnitts erwähnte Dienstprogramm 'make' basiert auf diesen Regeln. Auch Compiler und Linker greifen beim Auflösen von Importen darauf zurück.
Moderne Betriebssysteme fordern oder erlauben innerhalb eines Fileverzeichnisses Filenamen der Gestalt

bezeichner "." kennung,

wobei es oft für beide Komponenten Längenbeschränkungen gibt. Gängige Maximallängen für den 'bezeichner' sind 6, 8, 9 und 31. Als 'kennung' sind drei Zeichen gebräuchlich.
Die MODULA-2-Systeme gehen i. allg. davon aus, dass

der Modulname bzw. ein n Zeichen langes Anfangsstück des Modulnamens als 'bezeichner' verwendet wird und in der 'kennung' die unterschiedlichen "Filesorten" kodiert sind.

n ist die maximale 'bezeichner'-Länge. Folgende Kennungen sind üblich:

Inhalt der Files	'kennung'
Programm- und Implementationsmodule	MOD
Definitionsmodule	DEF
Symboltabellen	SYM (SFS, SFL)
Objektprogramme	LNK (OBJ)
Referenzen	REF
Protokolle	LST
Abarbeitungsfähige Programme	LOD

Der Programmierer braucht sich um diese Festlegungen nur bei der Erstellung seiner Quelltextdateien zu kümmern. Den Rest erledigt das MODULA-2-System. Wenn der Compiler auf eine den Baustein 'XYZ'

betreffende IMPORT-Klausel stösst, dann versucht er, das File 'XYZ.SYM' zu lesen. Findet er ein solches File nicht, dann wird i. allg. der Nutzer um die Eingabe eines Filenamens gebeten. In derselben Weise verfährt der Linker. Er würde das File 'XYZ.LNK' suchen.
Es ist unklug, die Namenskonvention nicht einzuhalten und sich auf die Möglichkeit der expliziten Namensangabe bei Anfragen zu verlassen. Wenn es Probleme mit der Eindeutigkeit der Filenamen gibt, dann sollte man den Modulnamen so "verbiegen", dass der abgeleitete Filename eindeutig ist.
Fortgeschrittene Systeme bieten Möglichkeiten zur Angabe spezieller Suchpfade für Files an. Grob gesagt geht es dabei um folgendes. Es wird festgelegt, in welchem Fileverzeichnis spezielle Files zu suchen sind. Auf diese Weise können z. B. die einzelnen Filesorten in getrennten Verzeichnissen aufbewahrt werden. Aber eines bleibt fest: Der Filename, der in den Verzeichnissen gesucht wird, ist nach den beschriebenen Regeln gebildet. Bisweilen kann man auch Alternativen formulieren: Wenn das File im ersten Verzeichnis nicht gefunden wurde, dann suche in einem zweiten usw. Näheres steht in der Nutzerdokumentation.
Viele Compiler und Linker nutzen die Namenskonvention für eine kleine Nutzerfreundlichkeit. Wenn man einen Filenamen angibt, der keinen Punkt enthält, dann ergänzt der Compiler diesen Namen vor der Suche mit ".MOD" und der Linker mit ".LNK". Wenn Ihr MODULA-2-System so arbeitet, dann gibt es nur noch eine Situation, in der Sie einen Filenamen mit Punkt und Kennung eintippen müssen: beim Übersetzen eines Definitionsmoduls.

10.4. Beispiel: Sortieren in Arrays

Viele Informationsverarbeitungsprozesse erfordern geordnete Datenmengen. Sortieren ist deshalb eine Standardaufgabe. Ich setze voraus, dass die zu sortierende Datenmenge aus Datensätzen eines festen Typs besteht und im Hauptspeicher meines Rechners Platz findet. Bei diesem Beispiel entstehen nacheinander zwei Implementationen für denselben Baustein sowie eine "Stopuhr" und ein Pseudozufallszahlengenerator.

Programm 10.1 zeigt den Definitionsmodul eines Sortierbausteins. Die zu sortierenden Datensätze sind vom Typ 'EINTRAG'. Das Sortiermerkmal - auch "Sortierschlüssel" genannt - befindet sich im Datenfeld 'Schluessel'. Der Einfachheit halber sehe ich dafür den Typ INTEGER vor. Der 'Inhalt' ist nur eine "Luftblase", deren Umfang durch die Konstante 'il' bestimmt wird. Ich empfehle Ihnen, nach dem Durcharbeiten des Beispiels eigene Laufzeitexperimente zu machen und den Einfluss der Datensatzgrösse auf die Sortierzeit zu untersuchen. Die Prozedur 'Sortiere' soll Anfangsstücke von Arrays des Typs 'FELD' nach aufteigenden Schlüsseln sortieren.

```
 1 DEFINITION MODULE Sortieren;
 2 (*-------------------------------------------------------------
 3   Sortieren von Datensaetzen in einem ARRAY
 4 -------------------------------------------------------------*)
 5 EXPORT QUALIFIED
 6   FeldLaenge, FELD, EINTRAG, Sortiere;
 7 (*-------------------------------------------------------------*)
 8 CONST
 9   il = 14; (* Das 'Schraeubchen' fuer Experimente mit unter-
10               schiedlich langen Datensaetzen                 *)
11 TYPE
12   EINTRAG = RECORD
13               Schluessel: INTEGER;
14               Inhalt: ARRAY [1..il] OF CHAR;
15             END;
16 CONST
17   FeldLaenge = 1000;
18 TYPE
19   FELD = ARRAY [0..FeldLaenge-1] OF EINTRAG;
20
21 PROCEDURE Sortiere(VAR f: FELD; bis: INTEGER);
22 (* Sortieren des Arrays 'f'
23    Nur die Elemente f[0], f[1], ..., f[bis] sind belegt.     *)
24 (*-------------------------------------------------------------*)
25 END Sortieren.
```

Programm 10.1. Definitionsmodul 'Sortieren'

An dem als Programm 10.2 abgedruckten Sortierverfahren ist für Sie höchstens der Name "Sortieren durch Auswählen" neu. Man sucht den 'EINTRAG' mit dem kleinsten 'Schluessel' heraus (Zeilen 11 bis 16) und vertauscht ihn mit dem ersten (Zeile 17). Dann beginnt die Sache von neuem, aber erst hinter dem eben auf seinen rechten Platz beförderten 'EINTRAG'. Schluss ist, wenn das "Reststück" nur noch aus der einzelnen Arraykomponente 'f[bis]' besteht. In den Programmen 2.2 und 9.2 wurde beim geordnete Ausschreiben ein ähnliches Prinzip verfolgt.

```
 1 IMPLEMENTATION MODULE Sortieren;
 2 (*-------------------------------------------------------------
 3    Sortieren von Datensaetzen in einem ARRAY (aufsteigend)
 4      METHODE: direktes Auswaehlen
 5 -------------------------------------------------------------*)
 6 PROCEDURE Sortiere(VAR f: FELD; bis: INTEGER);
 7 VAR
 8   i,j,min,minIndex: INTEGER; h: EINTRAG;
 9 BEGIN
10   FOR i:=0 TO bis-1 DO
11     min:=f[i].Schluessel; minIndex:=i;
12     FOR j:=i+1 TO bis DO
13       IF f[j].Schluessel<min THEN
14         min:=f[j].Schluessel; minIndex:=j;
15       END;
16     END;
17     h:=f[i]; f[i]:=f[minIndex]; f[minIndex]:=h;
18   END;
19 END Sortiere;
20 (*-------------------------------------------------------------*)
21 END Sortieren.
```

Programm 10.2. Sortierverfahren "Sortieren durch Auswählen"

Zur Erprobung des Bausteins 'Sortieren' habe ich ein Rahmenprogramm 'SortExperimente' geschrieben (Programm 10.3). Der Nutzer wird gefragt: "Wieviel Datensaetze soll ich sortieren?" Die Antwort "0" beendet das Programm. Ein "Sortierexperiment" besteht aus vier

Aufrufen der Prozedur 'Sortiere'. Nach einer "zufälligen" Folge von Schlüsselwerten kommen drei Extremfälle: inverse Schlüsselfolge, sortierte Schlüsselfolge und gleiche Schlüssel.

```
 1 MODULE SortExperimente;
 2 (*-------------------------------------------------------------
 3    Rahmen fuer Experimente mit dem Baustein 'Sortieren'
 4 -------------------------------------------------------------*)
 5 FROM Sortieren IMPORT FeldLaenge, FELD, Sortiere;
 6 FROM Fragen IMPORT ZahlFrage;
 7 FROM ZufallsZahlenGenerator IMPORT RandomInt;
 8 FROM InOut IMPORT WriteInt, WriteString, WriteLn;
 9 (*-------------------------------------------------------------*)
10 VAR
11   a: FELD;  n,i,Q: INTEGER;
12
13 PROCEDURE Aktion(txt: ARRAY OF CHAR);
14 BEGIN
15   WriteString(txt);
16   Sortiere(a,n-1);
17   WriteString(" - Fertig"); WriteLn;
18 END Aktion;
19 (*-------------------------------------------------------------*)
20 BEGIN
21   LOOP ZahlFrage("Wieviel Datensaetze soll ich sortieren?",
22                  0,FeldLaenge, n);
23     IF n=0 THEN RETURN END;
24     Q:=1; FOR i:=0 TO n-1 DO a[i].Schluessel:=RandomInt(Q) END;
25     Aktion('Sortieren des "zufaellig" belegten Arrays');
26     FOR i:=0 TO n-1 DO a[i].Schluessel:=-i END;
27     Aktion("Sortieren des umgekehrt sortierten Arrays");
28     Aktion("Sortieren des sortierten Arrays          ");
29     FOR i:=0 TO n-1 DO a[i].Schluessel:=0 END;
30     Aktion("Sortieren gleicher Schluessel            ");
31   END;
32 END SortExperimente.
```

Programm 10.3. Test- und Experimentierrahmen zum 'Sortieren'

Die einzelnen Aufrufe von 'Sortiere' sind innerhalb der Prozedur 'Aktion' (Zeilen 13 bis 18) zwischen zwei Schreiboperationen eingebettet: Man kann am Bildschirm genau die Dauer der Sortiervorgänge beobachten.
Die "zufälligen" Schlüsselfolgen hole ich mir von einem Baustein 'ZufallsZahlenGenerator', dessen selbsterklärender Definitionsmodul als Programm 10.4 abgedruckt ist.

```
 1 DEFINITION MODULE ZufallsZahlenGenerator;
 2 (*-------------------------------------------------------------
 3   Erzeugung von Folgen gleichverteilter Pseudozufallszahlen
 4   mit einem gemischten Kongruenzgenerator
 5   Periodenlaenge: 2**16
 6 -------------------------------------------------------------*)
 7 EXPORT QUALIFIED RandomInt, Random;
 8 (*-------------------------------------------------------------*)
 9 PROCEDURE RandomInt(VAR quelle: INTEGER): INTEGER;
10   (* Im Parameter 'quelle' schreibt der Generator den Stand
11      der Erzeugung der Folge fort. Die als Parameter ueber-
12      gebene Variable muss jeweils vor dem ersten Aufruf be-
13      legt werden.                                          *)
14 PROCEDURE Random(): INTEGER;
15   (* Wie 'RandomInt', aber die 'quelle' wird intern gehalten
16      und ist vor dem ersten Aufruf mit dem Wert 1 belegt.  *)
17 (*-------------------------------------------------------------*)
18 END ZufallsZahlenGenerator.
```

Programm 10.4. Baustein zur Erzeugung von Pseudozufallszahlen

Im Programmodul 'SortExperimente' wird die INTEGER-Variable 'Q' als 'quelle' für den Generator 'RandomInt' benutzt. Wegen der Wertzuweisung am Anfang von Zeile 24 entsteht bei jedem Sortierexperiment ein Anfangsstück derselben Folge von Pseudozufallszahlen. Wem das nicht gefällt, der mag die Anweisung 'Q:=1;' aus der LOOP-Anweisung herausziehen und sie gleich hinter BEGIN (Zeile 20) notieren.
Ursprünglich enthielt der Programmodul auch noch Prozeduren zum Ausschreiben der Schlüssel und zur Prüfung, ob die Schlüsselfolge wirklich aufsteigend geordnet ist. Derlei können Sie nach Bedarf

wieder einbauen und an geeigneten Stellen von 'Aktion' aufrufen. Wenn Sie ein wenig mit dem Programm 'SortExperimente' herumprobieren, dann fällt Ihnen gewiss auf, dass die Sortierzeit mit zunehmender Länge der zu sortierenden Folgen merklich wächst. Was tun? Lesen! In der Fachliteratur (z. B. [HoSa81], [Wirt86]) sind viele Sortierverfahren beschrieben und bez. ihres Laufzeitbedarfs mit mathematischen Methoden analysiert worden. "Quicksort" [Hoar62] ist für unsere Aufgabe der Favorit (s. 'quick' im Programm 10.5).
Ich habe zum Vergleich des "Sortierens durch Auswählen" mit "Quicksort" auf einem IBM-PC/XT-kompatiblen Rechner folgende Zeiten gemessen (Angaben in Sekunden):

Sortierlänge	Auswählen	Quicksort
100	0,4	0,2
200	1,5	0,3
300	3,4	0,6
400	5,9	0,7
500	9,1	0,9
600	13,1	1,1
700	17,8	1,3
800	23,1	1,6
900	29,2	1,8
1000	36,0	2,1

Wichtig ist vor allem der wesentlich langsamere Anstieg der Zeiten in der rechten Spalte. Sortieren durch Auswählen benötigt für 1000 Sätze das 90fache der bei 100 Sätzen erforderlichen Zeit. Quicksort nur das 10fache.
Wie funktioniert die rekursive Sortierprozedur 'quick', die ein so erstaunliches Zeitverhalten aufweist? Sie bearbeitet bei jedem Aufruf ein zusammenhängendes Teilstück des Arrays 'f':

f[links], f[links+1], ..., f[rechts-1], f[rechts].

Zuerst wird ein Schlüsselwert herausgegriffen und zum Wert der Variablen 'x' gemacht (erste Anweisung auf Zeile 12). Anschliessend (Zeilen 13 bis 18) wird das interessierende Teilstück von 'f' in zwei Abschnitte geteilt. Im linken stehen Datensätze, deren Schlüssel kleiner oder gleich 'x' ist. Im rechten sind alle Schlüssel grösser oder gleich 'x'.

```
 1 IMPLEMENTATION MODULE Sortieren;
 2 (*-------------------------------------------------------------
 3    Sortieren von Datensaetzen in einem ARRAY (aufsteigend)
 4      METHODE: Quicksort
 5 -------------------------------------------------------------*)
 6 PROCEDURE Sortiere(VAR f: FELD; bis: INTEGER);
 7
 8   PROCEDURE quick(links,rechts: INTEGER);
 9   VAR
10     i,j,x: INTEGER; h: EINTRAG;
11   BEGIN
12     x:=f[(links+rechts) DIV 2].Schluessel; i:=links; j:=rechts;
13     REPEAT
14       WHILE f[i].Schluessel<x DO INC(i) END;
15       WHILE f[j].Schluessel>x DO DEC(j) END;
16       IF i<j THEN h:=f[i]; f[i]:=f[j]; f[j]:=h END;
17       IF i<=j THEN INC(i); DEC(j) END;
18     UNTIL i>j;
19     IF links<j THEN quick(links,j) END;
20     IF i<rechts THEN quick(i,rechts) END;
21   END quick;
22
23 BEGIN
24   quick(0,bis);
25 END Sortiere;
26 (*-------------------------------------------------------------*)
27 END Sortieren.
```

Programm 10.5. Sortierverfahren "Quicksort"

Zu diesem Zweck wird von 'links' beginnend das erste 'i' mit 'f[i].Schluessel>=x' gesucht (Zeile 14). Von 'rechts' suchen wir das erste 'j' mit 'f[j].Schluessel<=x' (Zeile 15). Nun vertauschen wir 'f[i]' mit 'f[j]' (Zeile 16), erhöhen 'i', verringern 'j' (Zeile 17) und suchen an beiden "Enden" weiter, bis diese Enden sich "getroffen" haben (Bedingung in Zeile 18).
Was wurde damit erreicht? Wir haben die Aufgabe "Sortieren der Elemente f[links] bis f[rechts]" auf zwei kleinere Sortieraufgaben

zurückgeführt: "Sortiere den linken Abschnitt" und "Sortiere den rechten Abschnitt"! Abschnitte der Länge 1 sind sortiert. Längere Abschnitte lassen wir auf den Zeilen 19 und 20 durch 'quick' bearbeiten. Da das Einteilen in die zwei Abschnitte die entscheidende Idee von Quicksort ist, nennt man dieses Verfahren bisweilen auch "Sortieren durch Einteilen".

Die erwähnte mathematische Analyse des Rechenzeitbedarfs der Sortierverfahren ([HoSa81], [Wirt86]) liefert für die Anzahl der zum Sortieren eines Arrays der Länge n notwendigen Vergleichsoperationen folgende Formeln, wobei ln(x) der natürliche und ld(x) der Logarithmis zur Basis 2 ist.

Sortieren durch Auswählen	(n*n-n)/2
Quicksort	n*ld(n)*2*ln(2) (Mittelwert)

Die zweite Funktion wächst wesentlich langsamer als die erste. Meine Messergebnisse widersprechen diesen theoretischen Resultaten nicht. Das heisst, dass meine Programme keine prinzipiellen Fehler enthalten. Beim Übergang von 100 zu 1000 ergeben sich nach den Formeln Faktoren von 100 und 15. Gemessen habe ich 90 und 10.

Der kitzlige Punkt bei Quicksort ist die Auswahl des Vergleichsschlüssels 'x' für die Einteilung (Zeile 12). Gleichberechtigt zu dem programmierten Griff in die Mitte könnte man für 'x' auch stets den Schlüssel des ersten Elements nehmen. Aber bei dieser Variante wird das Sortieren des bereits sortierten Arrays zum schlechtesten Fall.

Der bez. der Rechenzeit ungünstigste Fall von Quicksort liegt nämlich vor, wenn bei der Auswahl von 'x' immer der grösste oder der kleinste Schlüssel des betrachteten Arrays auftritt. Dann wird die Sortieraufgabe der Länge n in eine der Länge n-1 und eine der Länge 1 zerlegt usw. Ideal wäre eine zufällige Auswahl des Vergleichsschlüssels. Empfohlen wird auch, den mittleren von drei willkürlich herausgegriffenen Schlüsseln zu nehmen.

Bei der Arbeit am schlechtesten Fall tritt eine Verschachtelung der Aufrufe von 'quick' bis zur Tiefe n auf. Und das führt für ausreichend grosses n bald zum Programmabbruch wegen Speicherplatzmangels, es sei denn, man verfügt über ein System mit virtuellem Speicher.

Zur Klärung dieses Problems folgen wir zunächst einer unabhängigen anderen Idee. Man spart die "Hälfte" der rekursiven Aufrufe, wenn

an Stelle des Aufrufs 'quick(i,rechts)' (Zeile 20) eine Wertzuweisung 'links:=i' programmiert wird und die Zeilen 12 bis 20 durch LOOP-END geklammert werden. Als Bremse ist in Zeile 20 ein ELSE-Zweig mit einer EXIT- oder RETURN-Anweisung aufzunehmen.
Das Problem mit dem Speicherüberlauf wegen zu tiefer Verschachtelung der Prozeduraufrufe klärt sich durch eine kleine Modifikation dieser Variante:

> 'quick' wird nicht für den linken, sondern für den kürzeren der bei der Einteilung entstandenen Abschnitte rekursiv aufgerufen.

Die entsprechend umgeschriebene Prozedur 'quick' hat folgendes Aussehen. Veränderungen gegenüber Programm 10.5 sind fett gedruckt.

```
 1 PROCEDURE quick(links,rechts: INTEGER);
 2 VAR
 3   i,j,x: INTEGER; h: EINTRAG;
 4 BEGIN
 5   LOOP
 6     x:=f[(links+rechts) DIV 2].Schluessel; i:=links; j:=rechts;
 7     REPEAT
 8       WHILE f[i].Schluessel<x DO INC(i) END;
 9       WHILE f[j].Schluessel>x DO DEC(j) END;
10       IF i<j THEN h:=f[i]; f[i]:=f[j]; f[j]:=h END;
11       IF i<=j THEN INC(i); DEC(j) END;
12     UNTIL i>j;
13     IF j-links < rechts-i THEN
14       IF links<j THEN quick(links,j) END;
15       IF i<rechts THEN links:=i ELSE RETURN END;
16     ELSE
17       IF i<rechts THEN quick(i,rechts) END;
18       IF links<j THEN rechts:=j ELSE RETURN END;
19     END;
20   END;
21 END quick;
```

Ich habe zu Beginn des Sortierbeispiels empfohlen, eigene Laufzeitexperimente zu machen. Zeitmessungen mit der Stopuhr am Bildschirm sind gewiss ein Anachronismus: Im Rechner ist normalerweise ein Uhr eingebaut, die man benutzen kann.
Ich verwende für Laufzeituntersuchungen den Baustein 'ZeitMessung',

dessen Definitionsmodul als Programm 10.6 abgedruckt ist.

```
 1 DEFINITION MODULE ZeitMessung;
 2 (*-------------------------------------------------------------
 3    Ermitteln und Ausschreiben der Laenge von Zeitintervallen
 4 -------------------------------------------------------------*)
 5 EXPORT QUALIFIED Start, Stop;
 6 (*-------------------------------------------------------------*)
 7 PROCEDURE Start;
 8 (* Beginn der zu vermessenden Zeitintervalle merken          *)
 9 PROCEDURE Stop;
10 (* Ausschreiben der seit dem letzten 'Start' vergangenen
11    Zeit im Format:   z{z} ":" z z "," z   (z = ziffer)
12 -------------------------------------------------------------*)
13 END ZeitMessung.
```

Programm 10.6. Baustein für Laufzeitmessungen

Der Programmodul 'SortierExperimente' kann sehr leicht um den Zugriff auf die Uhr erweitert werden. Man importiere die Bezeichner 'Start' und 'Stop'. 'Start' muss innerhalb der Prozedur 'Aktion' unmittelbar vor dem Sortieren (Programm 10.3, Zeile 16) gerufen werden und 'Stop' gleich nach dem Sortieren. Das ist alles.
Die Implementierung des Bausteins 'ZeitMessung' greift auf einen Typ 'Time' und eine Prozedur 'GetTime' zurück, die an irgendeiner Stelle der zum MODULA-2-System gehörenden Bausteinbibliothek exportiert werden. Die Stelle ist nicht einheitlich. Bei mir heisst der entsprechende Baustein 'Clock'. Sein Definitionsmodul folgt als Programm 10.7. Zwei weitere Bausteinnamen sind als Kommentare vermerkt.

```
 1 DEFINITION MODULE Clock;
 2                (* TimeDate; (*  in [Logi85]  *)             *)
 3                (* System;   (*  in [Robo87b] *)             *)
 4 (*-------------------------------------------------------------
 5    Bereitstellen von Uhrzeit und Datum
 6 -------------------------------------------------------------*)
 7 EXPORT QUALIFIED Time, GetTime;
```

```
 8 (*-----------------------------------------------------------*)
 9 TYPE
10   Time = RECORD
11            day, minute, millisec: CARDINAL;
12          END;
13 (* 'day' enthaelt in den Bits 0..4   Tag des Monats ([1..31])
14                  in den Bits 5..8   Monat des Jahres ([1..12])
15                  in den Bits 9..15  Jahr-1900.
16    Anders ausgedrueckt:
17        day = ( (Jahr-1900)*20B + Monat )*40B + Tag
18    'minute' enthaelt    Stunde*60+Minute
19    'millisec' enthaelt  Sekunde*1000+Millisekunde,
20                            beginnend mit 0 bei jeder vollen
21                            Minute                          *)
22
23 PROCEDURE GetTime(VAR t: Time);
24 (*
25    Liefert in 't' das aktuelle Datum und die aktuelle Zeit
26 -----------------------------------------------------------*)
27 END Clock.
```

Programm 10.7. Baustein zum Zugriff auf die rechnerinterne Uhr

Der Einfachheit halber setzt die als **Programm 10.8** abgedruckte Realisierung des Bausteins 'ZeitMessung' voraus, dass innerhalb des zu ermittelnden Zeitintervalls kein Ereignis der Art "Mitternacht" auftritt. Die Variable 'startZeit' wird erstmals im Initialisierungsteil belegt (Zeile 40). Das vermeidet Konvertierungsprobleme beim versehentlichen Aufruf von 'Stop' ohne vorherigen 'Start'.

```
 1 IMPLEMENTATION MODULE ZeitMessung;
 2 (*-----------------------------------------------------------
 3    Ermitteln und Ausschreiben der Laenge von Zeitintervallen
 4 -----------------------------------------------------------*)
 5 IMPORT Clock;
 6 FROM InOut IMPORT Write, WriteInt;
 7 (*-----------------------------------------------------------*)
 8 VAR  startZeit: Clock.Time;
```

```
 9
10 PROCEDURE Start;
11 BEGIN
12   Clock.GetTime(startZeit);
13 END Start;
14
15 PROCEDURE Stop;
16 VAR t: Clock.Time;  Minuten,Sekunden,Zehntel,Millis: CARDINAL;
17 BEGIN
18   Clock.GetTime(t);
19   WITH startZeit DO
20     Minuten:=t.minute-minute;
21     IF t.millisec<millisec THEN
22       DEC(Minuten); Millis:=60000-millisec+t.millisec;
23     ELSE
24       Millis:=t.millisec-millisec;
25     END;
26   END;
27   Sekunden := Millis DIV 1000; Millis := Millis MOD 1000;
28   Zehntel := (Millis+50) DIV 100;
29   IF Zehntel=10 THEN
30     Zehntel:=0; INC(Sekunden);
31     IF Sekunden=60 THEN Sekunden:=0; INC(Minuten) END;
32   END;
33   WriteInt(Minuten,2); Write(":");
34   IF Sekunden>=10 THEN WriteInt(Sekunden,2);
35   ELSE Write("0"); WriteInt(Sekunden,1) END;
36   Write(","); WriteInt(Zehntel,1);
37 END Stop;
38 (*-------------------------------------------------------------*)
39 BEGIN
40   Clock.GetTime(startZeit);
41 END ZeitMessung.
```

Programm 10.8. Implementation der Intervallzeitmessung

Als Abschluss des Beispiels "Sortieren" gebe ich eine Implementierung des Bausteins 'ZufallsZahlenGenerator' an. Aus der Literatur

ist bekannt, dass die Werte der CARDINAL-Variablen 'Quelle' beim wiederholten Ausführen der "Ergibtanweisung"

```
Quelle := (Quelle*31413 + 13849) MOD 2**16
```

alle Zahlen von 0 bis 2**16-1 in recht ungeordneter Weise durchlaufen (s. [Ahre85], [Ziel78]). Damit sie schnell sind, werden solche Generatoren meist sehr maschinennah programmiert. Bei dem hier verfolgten Zweck spielt jedoch die Geschwindigkeit praktisch keine Rolle. Ich realisiere den Generator also in MODULA-2.
Viele MODULA-2-Systeme brechen die Arbeit wegen CARDINAL overflow ab, wenn das Resultat einer arithmetischen Operation grösser als MAX(CARDINAL) ist. Die Funktionsprozedur 'Plus' auf den Zeilen 10 bis 19 von Programm 10.9 berechnet den Wert (a+b) MOD 2**16, ohne dass ein CARDINAL-Überlauf auftritt. Ich will auch x*y MOD 2**16 überlauffrei berechnen. Zur Vereinfachung der Schreibweise bei den folgenden Erläuterungen sei der Operator @ als

```
x @ y = (x+y) MOD 2**16
```

definiert. Weiter setze ich

```
x1 = x DIV 256  und  x2 = x MOD 256  (für y analog)
```

und erhalte die Gleichungskette

```
x*y MOD 2**16 = (x1*256 + x2) * (y1*256 +y2) MOD 2**16
    = (x1*x2*2**16 + (x1*y2 + x2*y1)*256 + x2*y2) MOD 2**16
    = (x1*y2 + x2*y1)*256 MOD 2**16  @  x2*y2
    = ((x1*y2 @ x2*y1) MOD 256) * 256  @  x2*y2.
```

Alle in der letzten Zeile auftretenden Multiplikationen sind sicher bez. CARDINAL-Überlauf. Auf den Zeilen 28 und 29 ist die Multiplikation von 'Quelle' mit 'Faktor' nach dieser Methode notiert. Die Addition von 'Summand' habe ich einbezogen. In 'RandomInt' geschieht dasselbe.

```
 1 IMPLEMENTATION MODULE ZufallsZahlenGenerator;
 2 (*-------------------------------------------------------------
 3   Erzeugung von Folgen gleichverteilter Pseudozufallszahlen
 4   mit einem gemischten Kongruenzgenerator
 5 -------------------------------------------------------------*)
 6 CONST       (* Fuer Periodenlaenge 2**16 *)
 7   Faktor = 31413 (* = F1*256 + F2 *);
 8   F1 = Faktor DIV 256; F2 = Faktor MOD 256;
 9   Summand = 13849;
```

```
10 PROCEDURE Plus(a,b: CARDINAL): CARDINAL;
11 CONST MAXCARDINAL = 65535;
12 VAR  x: CARDINAL;
13 BEGIN
14   IF a=0 THEN RETURN b END;
15   x := MAXCARDINAL-a+1;
16   IF x>b THEN RETURN a+b END;
17   IF x=b THEN RETURN 0 END;
18   RETURN b-x;
19 END Plus;
20 VAR
21   Quelle: CARDINAL; (* Fuer 'Random', Anfangswert: 1 *)
22 (*-------------------------------------------------------------*)
23 PROCEDURE Random(): INTEGER;
24 VAR Q1,Q2: CARDINAL;
25 BEGIN
26 (* Quelle := (Quelle*Faktor+Summand) (* MOD 2**16 *); *)
27   Q1 := Quelle DIV 256; Q2 := Quelle MOD 256;
28   Quelle := Plus( Plus(F1*Q2,F2*Q1) MOD 256 * 256,
29                   Plus(F2*Q2, Summand)            );
30   RETURN INTEGER(Quelle);
31 END Random;
32
33 PROCEDURE RandomInt(VAR quelle: INTEGER): INTEGER;
34 VAR Q1,Q2: CARDINAL;
35 BEGIN
36   Q1 := CARDINAL(quelle) DIV 256;
37   Q2 := CARDINAL(quelle) MOD 256;
38   quelle := INTEGER(Plus( Plus(F1*Q2,F2*Q1) MOD 256 * 256,
39                           Plus(F2*Q2, Summand)            ));
40   RETURN quelle;
41 END RandomInt;
42 (*-------------------------------------------------------------*)
43 BEGIN
44   Quelle := 1;
45 END ZufallsZahlenGenerator.
```

Programm 10.9. Ein langsamer Generator für Pseudozufallszahlen

11. Rechnen mit REAL-Zahlen

Der vorliegende Abschnitt kann keine Einführung in die "Numerische Mathematik" sein. Er will aber zeigen, wo die Ursachen mannigfacher Probleme liegen, und Denkanstösse vermitteln.
Am Anfang steht die Erläuterung der rechnerinternen Darstellung des Datentyps REAL und abgeleiteter "numerischer Effekte". Prozeduren zur Eingabe- und Ausgabekonvertierung von Zahlen kauft man meist "im Sack". Ich drucke einen durchschaubaren Baustein vollständig ab.

11.1. Computerzahlen

Der MODULA-2-Datentyp REAL wird durch die auf dem jeweiligen Computer verfügbaren Computerzahlen (man sagt auch: Gleitpunkt- oder Gleitkommazahlen) realisiert.
Definition (vgl. [KiSc88]):

Es seien b, l, e1 und e2 positive ganze Zahlen mit b>=2.
Die Menge R=R(b,l,e1,e2) der Computerzahlen zur Basis b mit der Mantissenlänge l und dem Exponentenbereich [-e1..e2] besteht aus allen Zahlen z der Gestalt

```
z = m * b**e.
```

e - der Exponent von z - ist eine beliebige ganze Zahl mit

```
-e1 <= e <= e2 oder e = e0 (Nullexponent).
```

m - die Mantisse von z - ist eine beliebige l-stellige gebrochene Zahl zur Basis b der Form

```
m = +/- m1 . m2 ... ml
  = +/- (m1 + m2*b**(-1) + ... + ml*b**(-l+1))
```

mit den Ziffern mi IN {0..b-1} (i=1,2,...,l), wobei entweder m1>=1 oder m1=m2=...=ml=0 und e=e0 gilt.

Die Forderung, dass bei Zahlen, die ungleich null sind, die erste Mantissenstelle verschieden von null sein muss, wird Normalisierungsbedingung genannt. Sie erzwingt die Eindeutigkeit der Zahldarstellung. Man spricht von normalisierten Gleitkommazahlen. Der Nullexponent e0 genügt i. allg. der Ungleichung e0<=-e1. Als Basis werden Zweierpotenzen bevorzugt. Üblich sind b=16 und b=2.

Im MODULA-2-System [Robo87b] wird der Typ REAL durch 32-Bit-Werte mit folgender Struktur dargestellt, wobei die Basis gleich 2 ist:

Bit 31 Vorzeichen (0 positiv, 1 negativ)
Bit 30-23 Charakteristik (= Exponent+127)
Bit 22-0 Mantissenstellen m2 bis m24.

Die Speicherung einer Charakteristik, die sich aus dem Exponenten und einem festen Zuschlag (hier 127) zusammensetzt, ist üblich. In diesem Zusammenhang ist es ausserdem üblich, den Nullexponenten e0 so festzulegen, dass sich dafür die Charakteristik 0 ergibt, und den Exponentenbereich so zu wählen, dass die Charakteristik bei von null verschiedenen Zahlen ungleich 0 ist.
Wo ist die Ziffer m1? Sie wurde "wegrationalisiert"! Wegen der Normalisierungsbedingung ist sie ausser bei der Null immer gleich 1. Da die Null an der Charakteristik 0 erkennbar ist, braucht das Bit m1 gar nicht mit abgespeichert zu werden.
Aus dem Gesagten und einigen wenigen Zusatzinformationen in der Nutzerdokumentation folgt, dass der Typ REAL in [Robo87b] durch die Computerzahlen R(2,24,126,127) mit e0=-127 realisiert ist. Als LONGREAL (manchmal auch als REAL) werden häufig die Computerzahlen R(2,53,1022,1023) benutzt. Man speichert in 8 Bytes ein Vorzeichenbit, eine 11-Bit-Charakteristik und die letzten 52 Mantissenstellen. Für die betragskleinste und -grösste Zahl ergeben sich folgende Werte:

R(2,24, 126, 127)	1.2E-38	3.4E38
R(2,53,1022,1023)	2.2E-308	1.8E308

Gegeben seien die Computerzahlen R = R(b,l,e1,e2). R ist eine endliche Menge. Für jeden Wert z aus R gilt die Ungleichung

-MAX < z < MAX mit MAX = b**(e2+1).

Wenn bei Operationen mit REAL-Zahlen Resultate auftreten, deren Betrag grösser oder gleich MAX ist, so spricht man von Überlauf (oder Exponentenüberlauf). Überlauf führt zum Programmabbruch und einer Ausschrift der Art floating point overflow. Dieses Problem gab es bei den INTEGER- und CARDINAL-Typen auch schon. Besonders lästig ist der Überlauf, wenn er bei Zwischenergebnissen auftritt, obwohl das Endresultat kleiner als MAX ist. Bisweilen helfen in solchen Situationen Skalierungsmassnahmen, deren Ziel darin besteht, weitgehend mit Zahlen zu rechnen, die kleiner als 1.0 sind.

Betrachten wir ein "klassisches" Beispiel! Die Länge der Diagonalen eines Rechtecks ergibt sich als Quadratwurzel der Summe der Seitenquadrate. Es kann sein, dass eines der Quadrate oder beide oder nur die Summe grösser als MAX sind, obwohl die Wurzel kleiner als MAX ist. Die naive "Geradeausberechnung" sqrt(a*a+b*b) versagt also. Wenn man vor dem Quadrieren mit dem Maximum der Werte von a und b skaliert (also durch diese Zahl dividiert) und die Wurzel hinterher entsprechend korrigiert, dann ist das Problem verschwunden. Das führt zur folgenden Prozedur 'diagonale'. Die Funktionsprozedur 'sqrt' ist vom Bibliotheksbaustein 'MathLib' (oder 'MathLib0') zu importieren.

```
PROCEDURE diagonale(a,b: REAL): REAL;
VAR h: REAL;
BEGIN
  IF a>b THEN h:=b/a; RETURN a*sqrt(h*h+1.0) END;
  h:=a/b; RETURN b*sqrt(h*h+1.0);
END diagonale;
```

Der zweite Effekt betrifft einen kleinen Bereich um die Null herum. Sei MIN = b**-e1. Es gibt genau eine Zahl z in R, die der Ungleichung -MIN<z<MIN genügt. Und das ist die Null. Wenn bei Operationen Resultate anfallen, die betragsmässig kleiner als MIN sind, so spricht man von Unterlauf (oder Exponentenunterlauf). Bei Unterlauf ist es üblich, dass der Computer als Resultat den Wert null liefert.

Ein dritter Effekt ist die Auslöschung. Er tritt bei der Subtraktion annähernd gleicher Zahlen auf. Die Darstellung beider Zahlen hat denselben Exponenten. Die vorderen Mantissenziffern sind gleich und heben sich auf (werden ausgelöscht). Beim Normalisieren kommt für jede vorn ausgelöschte Ziffer hinten eine Null in die Mantisse hinein. Diesen Nullen kann man nicht trauen.

Wenn z. B. bei dreistelliger Dezimalarithmetik R(10,3,25,25) die Werte 5.43*10**0 und 5.41*10**0 voneinander subtrahiert werden, dann ist das Ergebnis gleich 0.02*10**0 und nach der Normalisierung 2.00*10**-2. Dieses Resultat täuscht eine Genauigkeit vor, die wahrscheinlich durch nichts gerechtfertigt ist. Voraussetzung für die Exaktheit dieser Differenz wären nämlich auf fünf Stellen genaue Operanden.

Das theoretische Resultat einer Grundrechenoperation (+, -, *, /)

mit zwei Computerzahlen wird oft keine Computerzahl sein. Da auf dem Rechner als Ergebnis aber nur Computerzahlen möglich sind, muss zur nächstliegenden Computerzahl gerundet werden.
x und y seien zwei Computerzahlen. Wenn ich mit ex(x o y) das exakte und mit rd(x o y) das vom Computer ermittelte (also gerundete) Resultat der Operation x o y bezeichne, wobei o eine der Operationen Addition, Subtraktion, Multiplikation oder Division ist, dann gilt

rd(x o y) = ex(x o y) * (1-s), mit ABS(s) < mEps.

Dabei bezeichnet mEps die Maschinengenauigkeit b**-l. Das Runden geschieht natürlich automatisch. Aber man muss sich darüber im klaren sein, dass ein Resultat 1.0E14 bei Computerzahlen mit 24 Mantissenstellen nicht mehr bedeutet als die Information:

> Der "richtige" Wert liegt irgendwo maximal eine halbe Million von 10**14 entfernt.

Fassen wir zusammen: Rundungsfehler liegen bei der Arbeit mit Computerzahlen in der Natur der Sache. Sie können nicht vermieden werden. Jede Eingangsgrösse und alle Zwischenresultate sind damit behaftet.
Rundungsfehler pflanzen sich innerhalb einer Berechnung über die Zwischenergebnisse fort. Die Kunst des Programmierers besteht darin, ein Aufschaukeln dieser unvermeidlichen Fehler zu verhindern. Für die Lösung numerischer Aufgaben sucht man nach Berechnungsverfahren, bei denen die Rundungsfehler den Ablauf nicht zu sehr stören. Der auch heute noch empfehlenswerte "Klassiker" der einschlägigen Fachliteratur ist [Wilk69]. Theoretische Untersuchungen und Resultate findet man z. B. in [KiSc88]. [EnRe88] enthält eine interessante Sammlung von MODULA-2-Programmen für numerische Verfahren. Sehr viele praktische Hinweise und Anregungen bieten das Buch [Gand85] und die dazu erschienene Lösungssammlung [Gand86]. Dort wird auch das folgende Beispiel diskutiert.
Man sollte sich stets bemühen, Quellen für Auslöschungseffekte zu erkennen und zu vermeiden. Nach der Schulformel zum Lösen der quadratischen Gleichung x*x + p*x + q = 0 wäre etwa folgendes zu programmieren:

```
a := p/2.0; b := sqrt(a*a-q);
x1 := -a + b; x2 := -a - b;
```

Wenn ABS(p) verglichen mit ABS(q) sehr gross ist, dann tritt beim

Berechnen einer Wurzel Auslöschung auf, und diese Lösung wird ungenau. Der Effekt kann beseitigt werden, indem man die betragsmässig grössere Lösung nach der Schulformel ermittelt und daraus die zweite mittels des Viëtaschen Wurzelsatzes x1*x2=q gewinnt. Programm 11.1 kann zum Vergleich der mit dem naiven und dem verbesserten Verfahren erhaltenen Lösungen benutzt werden. Für ABS(p)>1.0 wurde gegen den Überlauf bei der Berechnung von (p/2.0)**2 eine Skalierung eingebaut. Man zieht p vor die Wurzel und erhält mit q/(p*p)=q/p/p den Ausdruck p*sqrt(0.25-q/p/p).

```
 1 MODULE QuadratischeGleichung;
 2 (*-------------------------------------------------------------
 3    Loesen der quadratischen Gleichung  x*x + p*x + q = 0.0
 4    Vergleich zweier Verfahren
 5 -------------------------------------------------------------*)
 6 FROM ZahlIO IMPORT ReadReal, WriteReal, inOrdnung;
 7 FROM InOut IMPORT Write, WriteString, WriteLn;
 8 FROM MathLib0 IMPORT sqrt;
 9 (*-------------------------------------------------------------*)
10 PROCEDURE qGlei1(p,q: REAL;
11                   VAR x1,x2: REAL;
12                   VAR komplex: BOOLEAN);
13 VAR d: REAL;
14 BEGIN
15   d:=p/2.0; d:=d*d-q;
16   komplex:=d<0.0; IF komplex THEN RETURN END;
17   d:=sqrt(d);
18   x1:=-h+d; x2:=-h-d;
19 END qGlei1;
20
21 PROCEDURE qGlei2(p,q: REAL; VAR x1,x2: REAL);
22 VAR d,f: REAL;
23 BEGIN
24   IF ABS(p)>1.0 THEN d:=0.25-q/p/p; f:=ABS(p);
25   ELSE d:=p/2.0; d:=d*d-q; f:=1.0 END;
26   x1 := ABS(p)/2.0 + f*sqrt(d); IF p>0.0 THEN x1:=-x1 END;
27   IF x1=0.0 THEN x2:=0.0 ELSE x2:=q/x1 END;
28 END qGlei2;
```

```
 29 (*-------------------------------------------------------------*)
 30 PROCEDURE R(z: CHAR; VAR v: REAL);
 31 BEGIN
 32   REPEAT
 33     Write(z); WriteString(" eingeben: "); ReadReal(v); WriteLn;
 34   UNTIL inOrdnung;
 35 END R;
 36
 37 PROCEDURE W(txt: ARRAY OF CHAR; x1,x2: REAL);
 38 BEGIN
 39   WriteString("Loesungen ("); WriteString(txt); Write(')');
 40   WriteReal(x1, 20,7, TRUE); WriteReal(x2, 20,7, TRUE);
 41   WriteLn;
 42 END W;
 43
 44 VAR
 45   p,q,x1,x2: REAL;
 46   komplex: BOOLEAN;
 47 (*-------------------------------------------------------------*)
 48 BEGIN
 49   WriteString
 50     ("Loesen der quadratischen Gleichung  x*x + P*x + Q = 0");
 51   WriteLn; R('P',p); R('Q',q);
 52   qGlei1(p,q, x1,x2, komplex);
 53   IF komplex THEN
 54     WriteString("Komplexe Loesungen"); WriteLn; RETURN;
 55   END;
 56   IF ABS(x1)>ABS(x2) THEN W("naiv  ",x1,x2);
 57   ELSE W("naiv  ",x2,x1) END;
 58   qGlei2(p,q, x1,x2); W("stabil",x1,x2);
 59 END QuadratischeGleichung.
```

Programm 11.1. Wurzeln der Gleichung x*x + p*x + q = 0

Die Hilfsprozeduren 'R' und 'W' erleichtern die E/A-Operationen. Für die Eingabe und das Ausschreiben von REAL-Werten benutze ich den Baustein 'ZahlIO' aus Abschnitt 11.2. Komplexe Lösungen interessieren hier nicht. Sie werden beim naiven Verfahren 'qGlei1'

diagnostiziert, und die Bearbeitung bricht ab, ohne das verbesserte Verfahren 'qGlei2' zu rufen (Zeilen 53 bis 55). Zur besseren Vergleichbarkeit der Ergebnisse schreibe ich auch beim naiven Verfahren die betragsgrössere Lösung zuerst aus (Zeilen 56 und 57).
Besonders deutlich wird der Auslöschungseffekt beim naiven Verfahren, wenn q=1.0 und p>2.0*b**((1+1) DIV 2) gewählt werden: x2=0.0!

Die Computerzahlen liegen zwischen -MAX und +MAX unterschiedlich dicht. Sei x eine beliebige positive Computerzahl (aber nicht die allergrösste) mit der Darstellung x = b**e * m. Wenn wir zur nächstgrösseren Computerzahl übergehen wollen, so muss in der letzten Mantissenstelle 1 addiert werden. Diese 1 hat innerhalb der Mantisse den Faktor b**(1-1), also insgesamt den Faktor b**(e+1-1). Mit anderen Worten: Der Abstand von der betrachteten Zahl x zur nächstgrösseren ist b**(e+1-1). Eine handhabbare Abschätzung dieser Differenz erhält man leicht auf folgende Weise. Aus der Definition der Computerzahlen ergibt sich die Beziehung

m < b.

Multiplizieren mit b**e liefert

b**e * m < b**(e+1).

Links steht x. Multiplikation mit b**-1 führt auf

x * b**-1 < b**(e+1-1).

Rechts steht gerade der Abstand von x zum "Nachfolger". Wenn wir zusammenfassen und die negativen Zahlen einbeziehen, so lässt sich feststellen:

> Der Abstand benachbarter Computerzahlen ist stets grösser als das Produkt der betragskleineren Zahl mit b**-1.

Für die Zahlen R(2,24,126,127) haben Nachbarn der Grössenordnung 2**44 (etwa 10**14) einen Abstand von mehr als 2**20 (etwa eine Million).
Aus der Untersuchung des Abstands benachbarter Zahlen ergibt sich eine interessante Charakterisierung der bereits im Zusammenhang mit den Rundungsfehlern erwähnten Maschinengenauigkeit:

> Sie ist die kleinste positive Computerzahl, die bei der Addition zu 1.0 ein von 1.0 verschiedenes Resultat liefert.

Das folgende Programm 11.2 berechnet auf der Grundlage dieser Charakterisierung die Maschinengenauigkeit. Vorher wird noch der Wert MIN ermittelt.

```
 1 MODULE MaschinenGenauigkeit;
 2 (*-------------------------------------------------------------
 3    Berechnen der Maschinengenauigkeit und der kleinsten
 4    darstellbaren positiven Zahl
 5 -------------------------------------------------------------*)
 6 FROM ZahlIO IMPORT WriteReal;
 7 FROM InOut IMPORT WriteString, WriteLn;
 8 (*-------------------------------------------------------------*)
 9 VAR
10   mEps, mMIN, eins, h: REAL;
11 BEGIN
12   h:=1.0;
13   REPEAT
14     mMIN:=h; h:=h/2.0;
15   UNTIL h=0.0;
16   WriteString("Kleinste positive Zahl mMIN =");
17   WriteReal(mMIN, 20,9, TRUE); WriteLn;
18   mEps:=1.0; eins:=1.0;
19   WHILE 1.0+mEps#eins DO mEps:=mEps/2.0 END;
20   WriteString("Maschinengenauigkeit   mEps =");
21   WriteReal(2.0*mEps, 20,9, TRUE); WriteLn;
22 END MaschinenGenauigkeit.
```

Programm 11.2. Ermitteln der Maschinengenauigkeit mEps und der kleinsten darstellbaren positiven Zahl mMIN

Die kleinste positive Computerzahl MIN ist eine Zweierpotenz. Also genügt es, 1.0 fortlaufend durch 2.0 zu dividieren, bis das Ergebnis 0.0 ist (Zeilen 12 bis 15). Unmittelbar davor hatte man MIN. Die Berechnung von 'mEps' ist analog. Die REAL-Variable 'eins' soll mögliche Optimierungen des Ausdrucks 1.0+mEps#1.0 unterbinden. Ein ausgefeilter Compiler würde diesen Vergleich zu mEps#0.0 umformen. Abschliessend gebe ich ein "albernes" Programm an. Es ist bekannt, dass die "harmonische Reihe"

$$1 + 1/2 + 1/3 + 1/4 + \ldots$$

divergiert, also über alle Grenzen wächst. Was geschieht auf dem Computer? Programm 11.3 berechnet die harmonische Reihe und gibt nach jeweils 10000 Summanden einen "Streckenbericht" aus.

Gleichzeitig wird geprüft, ob die Berechnung "festgefahren" ist.

```
 1 MODULE HarmonischeReihe;
 2 (*-------------------------------------------------------------
 3   Berechnung der harmonischen Reihe - Aus welchem Grund
 4   wird die Ausfuehrung des Programms abgebrochen?
 5 -------------------------------------------------------------*)
 6 FROM ZahlIO IMPORT WriteReal;
 7 FROM InOut IMPORT WriteString, WriteLn;
 8 (*-------------------------------------------------------------*)
 9 VAR
10   summe,s,n,nn: REAL;  i: INTEGER;
11
12 PROCEDURE Schreiben;
13   VAR ok: BOOLEAN;
14   BEGIN
15     ok:=WriteReal(n,16,6,TRUE) AND WriteReal(summe,16,6,TRUE);
16     WriteLn;
17   END Schreiben;
18
19 BEGIN
20   summe:=0.0; s:=summe; n:=0.0; nn:=n; i:=1;
21   LOOP
22     n:=n+1.0; summe:=summe+1.0/n;
23     IF i<10000 THEN INC(i) ELSE
24       i:=1; Schreiben;
25       IF summe#s THEN s:=summe ELSE
26         WriteString("Konvergenz!"); WriteLn; EXIT;
27       END;
28       IF n#nn THEN nn:=n ELSE
29         WriteString("n ist festgefahren!"); WriteLn; EXIT;
30       END;
31     END;
32   END;
33   Schreiben;
34 END HarmonischeReihe.
```

Programm 11.3. Berechnen der divergenten harmonischen Reihe

Wie beendet dieses Programm seine Arbeit? Vier Varianten sind denkbar:

1. Abbruch wegen Überlauf bei 'summe' (Zeile 22)
2. Abbruch wegen Überlauf bei 'n' (Zeile 22)
3. Unterlauf bei 1.0/n (Bremse auf den Zeilen 28 bis 30)
4. keine Veränderung mehr bei 'summe' auf Zeile 22 (Bremse auf den Zeilen 25 bis 27).

Überlegen Sie sich vor einem praktischen Experiment das Ergebnis!

11.2. Zahlkonvertierungen

Bausteine zur Konvertierung zwischen Zahlen und Zeichenketten bilden den Hintergrund jeglicher E/A-Operationen für Zahlen. Kann man sich auf die ausgeschriebenen Ergebnisse wirklich verlassen? Innerhalb von Konvertierungsprozeduren für REAL-Zahlen treten doch auch Rundungseffekte auf.

Genaugenommen ist es verwunderlich, dass man nach dem Einlesen der Zahl 0.1 beim Ausschreiben auch wieder den Wert 0.1 bekommt. Rechnerintern wird mit Dualbrüchen gearbeitet. Wenn man im Zahlensystem zur Basis 2 die Aufgabe 1/10 löst, dann ergibt sich ein unendlicher periodischer Dualbruch.

0.000 1100 1100 ...

Im Rechner wird er auf die entsprechende Computerzahl gerundet. Bei der Ausgabe wird diese Zahl wieder in einen Dezimalbruch zurückübersetzt. Auch hier muss gerundet werden.

Die meisten Konvertierungsprozeduren erzeugen bei der Umwandlung von REAL-Zahlen in Zeichenketten so viele Ziffern, wie man verlangt. Auf diese Weise wird eine nicht vorhandene Genauigkeit vorgetäuscht. Betrachten wir als Beispiel die Computerzahlen R(2,24,126,127). Es gibt nur 2**24 unterschiedliche Mantissen. Zur Darstellung dieser Mantissen im Dezimalsystem benötigt man maximal 8 Ziffern; denn der Wert 2**24 ist etwas grösser als 16 Millionen. Wenn ich eine Computerzahl als Dezimalzahl ausschreibe, dann kann ich mich also immer auf 7 und manchmal auch auf 8 Ziffern verlassen.

Programm 11.4 ist der Definitionsmodul eines Bausteins für Zahlkonvertierungen. Die dort eingeführte Prozedur 'RealInKette' nimmt

Bezug auf die der jeweiligen REAL-Darstellung entsprechende Maximalzahl signifikanter Dezimalstellen: Konstante 'MsDs' in Zeile 10 des zugehörigen Implementationsmoduls (Programm 11.5). Werden mehr Stellen verlangt, so wird auf 'MsDs' Stellen gerundet, und die restlichen Stellen erscheinen als kleines "o", damit man sie als nichtsignifikante Nullen erkennt. Der Definitionsmodul 'ZahlKonvertierung' enthält alle nötigen Angaben als Kommentare. Alle Prozeduren bearbeiten Teilstücke grösserer CHAR-Arrays. Sie können deshalb sehr flexibel einsetzbar.

```
 1 DEFINITION MODULE ZahlKonvertierung;
 2 (*-------------------------------------------------------------
 3   Konvertierungsprozeduren zwischen Zeichenketten und
 4   Zahlen in beiden Richtungen
 5 -------------------------------------------------------------*)
 6 EXPORT QUALIFIED
 7   KetteInInteger, KetteInReal,
 8   IntegerInKette, RealInKette;
 9 (*-------------------------------------------------------------
10   Die Prozeduren 'KetteInInteger' und 'KetteInReal' wandeln
11   Zeichenketten, die Zahlen repraesentieren, in Zahlen um.
12   PARAMETER:
13     k     - Zeichenkette
14     i     - Anfangs: Index des ersten zu betrachtenden Zeichens
15             Nach erfolgreicher Konvertierung enthaelt 'i' den
16             Index des ersten nichtverarbeiteten Zeichens (ggf.
17             gleich HIGH(s)+1). Wenn Konvertierungsfehler auftre-
18             ten, dann bleibt der Wert von 'i' erhalten.
19     e     - Index des letzten zu betrachtenden Zeichens
20     leer  - 'TRUE'   Fuehrende Leerzeichen und Tabulatoren sind
21                      erlaubt.
22             'FALSE': Beides ist verboten.
23     wert  - die konvertierte Zahl; bei Konvertierungsfehlern
24             jedoch null.
25   RESULTAT:
26     Wenn die Konvertierung moeglich war (richtiges Format und
27     keine Zahlbereichsueberschreitung), dann 'TRUE'. Andern-
28     falls 'FALSE'.
```

```
29   Beide Prozeduren erwarten am Anfang des Teilstuecks
30                k[i], k[i+1], ..., k[e]
31   der Zeichenkette 'k' die Dezimaldarstellung einer Zahl. Das
32   genaue Format ist bei den Prozedurkoepfen in EBNF angegeben.
33   Die Konvertierung stoppt beim ersten nicht zum jeweiligen
34   Format passenden Zeichen, spaetestens jedoch nach der Ver-
35   arbeitung von 'k[e]'.
36 ------------------------------------------------------------*)
37 PROCEDURE KetteInInteger(k: ARRAY OF CHAR;
38                          VAR i: CARDINAL;
39                          e: CARDINAL;
40                          leer: BOOLEAN;
41                          VAR wert: INTEGER    ): BOOLEAN;
42 (* Datenformat:
43    ganze-zahl_mit_vorzeichen = ["+"|"-"] ganze_zahl.
44    ganze_zahl = ziffer{ziffer}.
45    ziffer = "0"|"1"|     |"9".                              *)
46
47 PROCEDURE KetteInReal(k: ARRAY OF CHAR;
48                       VAR i: CARDINAL;
49                       e: CARDINAL;
50                       leer: BOOLEAN;
51                       VAR wert: REAL       ): BOOLEAN;
52 (* Datenformat:
53    reelle_zahl_mit_vorzeichen = ["+"|"-"] reelle_zahl.
54    reelle_zahl = ziffer{ziffer} ["."{ziffer}] [skalierung]
55                | "." ziffer{ziffer} [skalierung]
56                | skalierung.
57    skalierung = ("E"|"e") ["+"|"-"] ziffer{ziffer}.
58    ziffer = "0"|"1"|     |"9".                              *)
59 (*-------------------------------------------------------------
60   Die Prozeduren 'IntegerInKette' und 'RealInKette' wandeln
61   Zahlen in eine Zeichenkettendarstellung dieser Zahlen um.
62   PARAMETER:
63     wert - die zu konvertierende Zahl
64     k    - Zeichenkettenvariable fuer den konvertierten 'wert'
65     a    - Anfang (Index des ersten zu belegenden Zeichens)
```

```
66      l     - Laenge des zu belegenden Bereichs von 'k'
67      g     - Genauigkeit (Anzahl der Dezimalstellen hinter dem
68              Dezimalpunkt)
69      expo - 'TRUE'   Ausgabeformat mit Exponententeil (normali-
70                      sierte Gleitpunktdarstellung)
71             'FALSE': Ausgabe als reiner Dezimalbruch (Fest-
72                      punktdarstellug)
73    RESULTAT:
74      Wenn die Konvertierung moeglich war und die erzeugte
75      Zahldarstellung nicht mehr als 'l' Zeichen beansprucht
76      und in 'k' ab Index 'a' noch Platz fuer 'l' Zeichen war,
77      dann 'TRUE'. Sonst 'FALSE' ('k' bleibt ungeaendert).
78    Beide Prozeduren tragen die konvertierte Zahldarstellung
79    rechtsbuendig in den Abschnitt
80                  k[a], k[a+1], ..., k[a+l-1]
81    der Zeichenkettenvariablen 'k' ein. Ein negatives Vorzeichen
82    steht unmittelbar vor der Zahl. Positive Vorzeichen werden
83    nicht erzeugt. Eine fuehrende Null tritt nur direkt vor dem
84    Dezimalpunkt und bei der Darstellung von null auf. Links
85    werden ggf. fuehrende Leerzeichen erzeugt. Die genauen For-
86    matbeschreibungen finden sich bei den Prozedurkoepfen. Die
87    Prozedur 'RealInKette' nimmt Bezug auf die der jeweiligen
88    rechnerinternen REAL-Darstellung entsprechende Maximalzahl
89    signifikanter Dezimalstellen (Konstante 'MsDs'). Werden mehr
90    als 'MsDs' Stellen verlangt, so wird auf 'MsDs' Stellen ge-
91    rundet, und die restlichen Stellen erscheinen als "spezielle"
92    Nullen ("o").
93  ------------------------------------------------------------*)
94  PROCEDURE IntegerInKette(wert: INTEGER;
95                           VAR k: ARRAY OF CHAR;
96                           a,
97                           l: CARDINAL         ): BOOLEAN;
98  (* Datenformat:
99     ganze_zahl = ["-"] ziffer{ziffer}.
100    ziffer = "0"|"1"|     |"9".                              *)
101
```

```
102 PROCEDURE RealInKette(wert: REAL;
103                       VAR k: ARRAY OF CHAR;
104                       a,
105                       l,
106                       g: CARDINAL;
107                       expo: BOOLEAN       ): BOOLEAN;
108 (* Datenformate:
109    festpunktzahl = ["-"] ziffer{ziffer} [dezimalbruch].
110    dezimalbruch = "." ziffer{ziffer}.
111    gleitpunktzahl =
112       ["-"] ziffer dezimalbruch "e" ("+"|"-") ziffer ziffer.
113    ziffer = "0"|"1"|     |"9".                              *)
114 (*-------------------------------------------------------------*)
115 END ZahlKonvertierung.
```

Programm 11.4. Ein Baustein zur Konvertierung von Zahlen

Der Implementationsmodul 'ZahlKonvertierung' (Programm 11.5) ist recht umfangreich. Alle Prozeduren enthalten an ihrem Anfang Tests, ob die Längenangaben mit den übergebenen Zeichenkettenparametern verträglich sind. Bei den '...InKette'-Prozeduren wird nach der internen Ermittlung des wirklich benötigten Stellenzahl geprüft, ob die gewünschte Länge 'l' ausreicht.
Die Prozeduren zur Konvertierung von INTEGER-Werten sind leicht durchschaubar. 'KetteInReal' verarbeitet die Ziffern vor dem Dezimalpunkt, die Ziffern dahinter und die Skalierung jeweils extra und "bastelt" daraus den endgültigen Wert zusammen.
'RealInKette' normalisiert den Wert auf eine Darstellung mit einer Ziffer vor dem Dezimalpunkt. Diese Ziffer kann mit 'TRUNC' separiert und ausgegeben werden. Wenn man nun mit 10.0 multiplizert, funktioniert dasselbe für die nächste Stelle. Ziffernberechnung, "Ausgabe" und Multiplikation mit 10.0 finden in der Prozedur 'ziffer' statt.

```
 1 IMPLEMENTATION MODULE ZahlKonvertierung;
 2 (*-------------------------------------------------------------
 3   Konvertierungsprozeduren zwischen Zeichenketten und
 4   Zahlen in beiden Richtungen
 5 -------------------------------------------------------------
 6   Die folgenden zwei Konstanten haengen ab von der auf dem
 7   Rechner verfuegbaren REAL-Arithmetik!
 8 -------------------------------------------------------------*)
 9 CONST
10   MsDs = 7; (* Anzahl der signifikanten Dezimalziffern *)
11   ExpL = 2; (* Exponentenlaenge bei Realkonvertierung in ein
12               Gleitpunktformat *)
13   (* Diese Werte sind geeignet fuer eine rechnerinterne REAL-
14      Darstellung mit 7 Bit fuer den Exponenten und 24 Bit fuer
15      die Mantisse *)
16 (*-------------------------------------------------------------*)
17 CONST
18   tab = 11C; (* Tabulator *)
19
20 PROCEDURE p10(x: REAL; n:INTEGER): REAL;
21 (* Resultat:    x * 10.0**n    *)
22 VAR
23   i: INTEGER; r: REAL;
24 BEGIN
25   IF n=0 THEN RETURN x END;
26   r:=x;
27   IF n>0 THEN
28     FOR i:=1 TO n DO r:=r*10.0 END;
29   ELSE
30     FOR i:=1 TO ABS(n) DO r:=r/10.0 END;
31   END;
32   RETURN r;
33 END p10;
34 (*-------------------------------------------------------------*)
35 PROCEDURE KetteInInteger(k: ARRAY OF CHAR;
36                           VAR i: CARDINAL;
37                           e: CARDINAL;
38                           leer: BOOLEAN;
```

```
39                          VAR wert: INTEGER): BOOLEAN;
40 VAR
41   ii: CARDINAL;  w: INTEGER;  ch: CHAR;  negativ: BOOLEAN;
42
43 PROCEDURE nch;
44   BEGIN
45     IF ii<=e THEN ch:=k[ii]; INC(ii) ELSE ch:=0C END;
46   END nch;
47
48 BEGIN
49   wert:=0; ii:=i; nch; w:=0;
50   IF (i<0)OR(i>HIGH(k)) OR (e<i)OR(e>HIGH(k)) THEN
51     (* Die Parameter sind leider inkonsistent *) RETURN FALSE;
52   END;
53   IF leer THEN
54     WHILE (ch=' ')OR(ch=tab) DO nch END;
55   END;
56   IF (ch='+')OR(ch='-') THEN negativ := ch='-'; nch;
57   ELSE negativ := FALSE END;
58   IF (ch<'0')OR(ch>'9') THEN
59     (* Ziffer erwartet *) RETURN FALSE;
60   END;
61   REPEAT (*Hier wird konvertiert*)
62    w := w*10 + INTEGER(ORD(ch)-ORD('0')); nch;
63   UNTIL (ch<'0')OR(ch>'9');
64   IF negativ THEN wert:=-w ELSE wert:=w END;
65   i:=ii; RETURN TRUE;
66 END KetteInInteger;
67
68 PROCEDURE KetteInReal(k: ARRAY OF CHAR;
69                       VAR i: CARDINAL;
70                       e: CARDINAL;
71                       leer: BOOLEAN;
72                       VAR wert: REAL       ): BOOLEAN;
73 VAR
74   ganz,bruch: REAL;  ii: CARDINAL;  gl,bl,expo: INTEGER;
75   ch: CHAR;  negativ,nurExpo: BOOLEAN;
76
```

```
 77 PROCEDURE nch;
 78   BEGIN
 79     IF ii<=e THEN ch:=k[ii]; INC(ii) ELSE ch:=0C END;
 80   END nch;
 81
 82 PROCEDURE ziffern(VAR w: REAL; VAR l: INTEGER);
 83   BEGIN
 84     w:=0.0; l:=0;
 85     WHILE (ch>='0')AND(ch<='9') DO
 86       w := w*10.0 + FLOAT(ORD(ch)-ORD('0')); INC(l); nch;
 87     END;
 88   END ziffern;
 89
 90 BEGIN
 91   wert:=0.0; ii:=i; nch;
 92   IF (i<0)OR(i>HIGH(k)) OR (e<i)OR(e>HIGH(k)) THEN
 93     (* Die Parameter sind leider inkonsistent *) RETURN FALSE;
 94   END;
 95   IF leer THEN
 96     WHILE (ch=' ')OR(ch=tab) DO nch END;
 97   END;
 98   IF (ch='+')OR(ch='-') THEN negativ := ch='-'; nch;
 99   ELSE negativ := FALSE END;
100 (*Jetzt beginnt die Konvertierung*)
101   ziffern(ganz,gl);
102   IF ch='.' THEN
103     nch; ziffern(bruch,bl);
104     IF gl+bl=0 THEN
105       (*Dezimalpunkt ohne Ziffern*) RETURN FALSE;
106     END;
107   ELSE
108     bruch:=0.0; bl:=0;
109   END;
110   nurExpo := (gl=0)AND(bl=0); IF nurExpo THEN ganz:=1.0 END;
111   IF (ch='e')OR(ch='E') THEN
112     IF NOT KetteInInteger(k,ii,e,FALSE,expo) THEN
113       (* Der Exponent fehlt hinter e/E *) RETURN FALSE;
114     END;
```

```
115
116   ELSE
117     IF nurExpo THEN
118       (* Ueberhaupt nichts Vernuenftiges *) RETURN FALSE;
119     END;
120     expo:=0;
121   END;
122 (*Berechnung der Zahl aus ihren drei Bestandteilen*)
123   ganz:=p10(ganz,expo);
124   bruch:=p10(bruch,expo-bl);
125   wert:=ganz+bruch;
126   IF negativ THEN wert:=-wert END; i:=ii; RETURN TRUE;
127 END KetteInReal;
129
130 PROCEDURE IntegerInKette(wert: INTEGER;
131                          VAR k: ARRAY OF CHAR;
132                          a,
133                          l: CARDINAL         ): BOOLEAN;
134 VAR
135   kk: ARRAY [0..31] OF CHAR; (* Das ist gewiss gross genug. *)
136   w,laenge,i: CARDINAL;
137 BEGIN
138   IF a+l-1>HIGH(k) THEN
139     (* Platz reicht nicht aus *) RETURN FALSE;
140   END;
141   w:=ABS(wert); laenge:=0;
142   REPEAT
143     kk[laenge] := CHR(w MOD 10 + ORD('0'));
144     w := w DIV 10; INC(laenge);
145   UNTIL w=0;
146   IF laenge>l THEN
147     (* Zahl zu lang *) RETURN FALSE;
148   END;
149   FOR i:=1 TO laenge DO k[a]:=' '; INC(a) END;
150   IF wert>=0 THEN k[a]:=' ' ELSE k[a]:='-' END; INC(a);
151   FOR i:=laenge-1 TO 0 BY -1 DO k[a]:=kk[i]; INC(a) END;
152   RETURN TRUE;
153 END IntegerInKette;
```

```
154
155 PROCEDURE RealInKette(wert: REAL;
156                       VAR k: ARRAY OF CHAR;
157                       a,
158                       l,
159                       g: CARDINAL;
160                       expo: BOOLEAN       ): BOOLEAN;
161 VAR
162   r: REAL;
163   ex,gg,ziffernVorPunkt,anzahl,i: INTEGER;
164   laenge,e: CARDINAL;
165
166 PROCEDURE ziffer;
167   VAR z: CARDINAL;
168   BEGIN
169     IF anzahl<MsDs THEN
170       z:=TRUNC(r); r:=(r-FLOAT(z))*10.0;
171       k[a]:=CHR(z+ORD('0'));
172     ELSE
173       k[a]:='o';
174     END;
175     INC(a); INC(anzahl);
176   END ziffer;
177
178 BEGIN
179   IF a+l-1>HIGH(k) THEN
180     (* Platz reicht nicht aus *) RETURN FALSE;
181   END;
182   r:=ABS(wert); ex:=0;
183   IF wert#0.0 THEN
184     (*Berechnung von 'r' und 'ex' mit
185         1.0 <=r < 10.0     und
186         r*10.0**ex = ABS(wert)                                    *)
187     IF r>=10.0 THEN
188       REPEAT r:=r/10.0; INC(ex) UNTIL r<10.0;
189     ELSE
190       WHILE r<1.0 DO r:=r*10.0; DEC(ex) END;
191     END;
```

```
192     (*Runden*)
193     gg:=g; IF NOT expo THEN INC(gg,ex) END;
194     IF gg>MsDs-1 THEN gg:=MsDs-1 END;
195     r := r + p10(1.0,-gg)/2.0;
196     IF r>=10.0 THEN r:=r/10.0; INC(ex) END;
197   END (*wert#0.0*);
198   IF expo OR (ex<0) THEN ziffernVorPunkt:=1;
199   ELSE ziffernVorPunkt:=1+ex END;
200   laenge:=1+ziffernVorPunkt; (* '-' oder   ' und Ziffern *)
201   IF g>0 THEN INC(laenge,1+g) (*     und Ziffern *) END;
202   IF expo THEN INC(laenge,2+ExpL) (* 'e', Vorz., Ziffern *) END;
203   IF laenge>l THEN
204     (* Zahl zu lang *) RETURN FALSE;
205   END;
206   FOR i:=1 TO l-laenge DO k[a]:=' '; INC(a) END;
207   IF wert>=0.0 THEN k[a]:=' ' ELSE k[a]:='-' END;
208   INC(a); anzahl:=0;
209   IF expo OR (ex>=0) THEN
210     FOR i:=1 TO ziffernVorPunkt DO ziffer END;
211   ELSE
212     k[a]:='0'; INC(a);
213   END;
214   IF g>0 THEN k[a]:='.'; INC(a) END;
215   IF NOT expo AND (ex<0) THEN
216     FOR i:=-2 TO ex BY -1 DO k[a]:='0'; INC(a) END;
217   END;
218   FOR i:=1 TO g DO ziffer END;
219   IF expo THEN
220     k[a]:='e'; INC(a);
221     IF ex>=0 THEN k[a]:='+' ELSE k[a]:='-' END;
222     INC(a); e:=ABS(ex);
223     FOR i:=a+ExpL-1 TO a BY -1 DO
224       k[i] := CHR(e MOD 10 + ORD('0')); e := e DIV 10;
225     END;
226   END;
227   RETURN TRUE;
228 END RealInKette;
```

```
229 (*-------------------------------------------------------------*)
230 END ZahlKonvertierung.
```

Programm 11.5. Konvertierung von Zahlen

Als naheliegende Anwendung des Bausteins 'ZahlKonvertierung' definiere ich im **Programm 11.6** einen Baustein zur Eingabe und Ausgabe von Zahlen. Die exportierte Variable 'inOrdnung' zeigt analog zu 'InOut.Done' an, ob die verlangte E/A-Operation erfolgreich abgeschlossen wurde.

```
 1 DEFINITION MODULE ZahlIO;
 2 (*-------------------------------------------------------------
 3   Prozeduren zur Eingabe und Ausgabe von Zahlen ueber das Ter-
 4   minal oder von/in Files
 5     - Datenformate und Parameter entsprechen den Prozeduren
 6       des Bausteins 'ZahlKonvertierung'.
 7     - Zum Lesen und Schreiben wird auf den Baustein 'InOut' zu-
 8       rueckgegriffen, d.h., Standardeingabe = Standardausgabe =
 9       Terminal; E/A-Umlenkungen sind durch 'InOut.OpenOutput'
10       und 'InOut.OpenInput' moeglich.
11 -------------------------------------------------------------*)
12 EXPORT QUALIFIED
13   ReadInteger, ReadReal, WriteInteger, WriteReal, inOrdnung;
14 (*-------------------------------------------------------------*)
15 VAR
16   inOrdnung: BOOLEAN;
17  (* Wird von allen Prozeduren gesetzt.
18       'TRUE'  - Operation ausgefuehrt
19       'FALSE' - sonst                                          *)
20
21 PROCEDURE ReadInteger(VAR wert: INTEGER);
22
23 PROCEDURE ReadReal(VAR wert: REAL);
24
```

```
25 PROCEDURE WriteInteger(wert: INTEGER;
26                        l: CARDINAL  );
27
28 PROCEDURE WriteReal(wert: REAL;
29                     l,
30                     g: CARDINAL;
31                     expo: BOOLEAN);
32 (*------------------------------------------------------------*)
33 END ZahlIO.
```

Programm 11.6. Ein Baustein zur Eingabe und Ausgabe von Zahlen

Dreh- und Angelpunkt der Relisierung des Bausteins 'ZahlIO' (**Programm 11.7**) ist die Variable 'puffer'. Bei Eingaben wird er mit 'InOut.ReadString' gefüllt. Bei Ausgaben wird er mit 'InOut.WriteString' ausgeschrieben. Den Rest erledigt die jeweilige Konvertierungsprozedur, deren Resultat direkt als Wert für die Anzeigevariable 'inOrdnung' verwendbar ist.

```
 1 IMPLEMENTATION MODULE ZahlIO;
 2 (*------------------------------------------------------------
 3   Prozeduren zur Eingabe und Ausgabe von Zahlen ueber das Ter-
 4   minal oder von/in Files
 5 ------------------------------------------------------------*)
 6 FROM ZahlKonvertierung IMPORT
 7   KetteInInteger, KetteInReal, IntegerInKette, RealInKette;
 8 FROM InOut IMPORT
 9   ReadString, WriteString;
10 (*------------------------------------------------------------*)
11 CONST
12   pl = 80; (* Laenge des internen Pufers, eine Zeile *)
13 VAR
14   puffer: ARRAY [0..pl-1] OF CHAR;
15
```

```
16 PROCEDURE InitPuffer;
17 VAR
18   i: CARDINAL;
19 BEGIN
20   FOR i:=0 TO pl-1 DO puffer[i]:=0C END;
21 END InitPuffer;
22

23 PROCEDURE ReadInteger(VAR wert: INTEGER);
24 VAR
25   i: CARDINAL;
26 BEGIN
27   InitPuffer; i:=0;
28   ReadString(puffer);
29   inOrdnung:=KetteInInteger(puffer,i,pl-1,TRUE,wert);
30 END ReadInteger;
31

32 PROCEDURE ReadReal(VAR wert: REAL);
33 VAR
34   i: CARDINAL;
35 BEGIN
36   InitPuffer; i:=0;
37   ReadString(puffer);
38   inOrdnung:=KetteInReal(puffer,i,pl-1,TRUE,wert);
39 END ReadReal;
40

41 PROCEDURE WriteInteger(wert: INTEGER;
42                         l: CARDINAL  );
43 BEGIN
44   inOrdnung:=IntegerInKette(wert,puffer,0,l);
45   IF inOrdnung THEN
46     IF l<pl THEN puffer[l]:=0C END; WriteString(puffer);
47   END;
48 END WriteInteger;
49
```

```
 50 PROCEDURE WriteReal(wert: REAL;
 51                     l,
 52                     g: CARDINAL;
 53                     expo: BOOLEAN);
 54 BEGIN
 55   inOrdnung:=RealInKette(wert,puffer,0,l,g,expo);
 56   IF inOrdnung THEN
 57     IF l<pl THEN puffer[l]:=0C END; WriteString(puffer);
 58   END;
 59 END WriteReal;
 60 (*-------------------------------------------------------------*)
 61 END ZahlIO.
```

Programm 11.7. Eingabe und Ausgabe von Zahlen

12. Pointertypen und dynamische Variablen

Pointertypen bilden die Basis für zwei interessante Programmierkonzepte:

1. Arbeit mit Zeigern auf Objekte
2. Erzeugen neuer Objekte, die in keiner 'variablenvereinbarung' erscheinen.

In MODULA-2 sind Punkt 1 und Punkt 2 fest miteinander verbunden. Einerseits können Zeiger nur auf dynamisch erzeugte Objekte verweisen. Andererseits sind solche Objekte nur über Zeiger "erreichbar". Zeigerwerte entstehen einzig und allein durch die für das Erzeugen eines neuen Objekts zuständige Operation 'NEW'. Es ist nicht möglich, mittels eines Zeigers auf eine in einer 'variablenvereinbarung' eingeführte Variable oder eine Komponente davon zu verweisen. Die in Variablenvereinbarungen eingeführten Objekte bezeichnet man als statische Variablen. Im Gegensatz dazu werden die durch 'NEW' erzeugten Objekte **dynamische Variablen** genannt.
Bild 12.1 zeigt eine oft verwendete Datenstruktur. An der statischen Zeigervariablen 'Anfang' hängt eine mit Hilfe von Zeigern aufgebaute Kette von dynamischen Variablen. Der Terminus technicus dafür lautet: "einfach verkettete Liste". Zeigerwerte sind als Pfeile dargestellt.

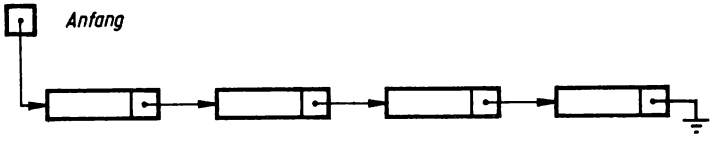


Bild 12.1. Eine dynamische Datenstruktur: einfach verkettete Liste

Die für Bereitstellung und Vernichtung dynamischer Variablen zuständigen Standardprozeduren greifen auf den Bibliotheksbaustein 'Storage' zurück. Er wird im Abschnitt 12.1 nach den Pointertypen und der Notation des Zugriffs auf dynamische Variablen behandelt. Abschnitt 12.2 befasst sich mit binären Bäumen, einer häufig einsetzbaren dynamischen Datenstruktur.

12.1. Pointertypen und der Bibliotheksbaustein 'Storage'

Die Vereinbarung eines Pointertyps legt fest, an welchen anderen Typ der Pointertyp "gebunden" ist.

```
$    pointertyp = POINTER TO typ.
```

Wenn 'T' ein Typ ist, dann beschreibt der Pointertyp

```
     POINTER TO T
```

Zeiger auf Variablen des Typs 'T'. Solche Zeiger werden auch Verweise oder Pointer genannt. Zu Bild 12.1 gehören die folgenden Vereinbarungen:

```
     TYPE
       eListe = POINTER TO eListenEintrag;
       eListenEintrag = RECORD
                          inhalt: INTEGER;
                          nachfolger: eListe;
                        END;
     VAR
       Anfang: eListe;
```

Die vier dickumrandeten Rechtecke im Bild 12.1 stellen dynamische Variablen des Typs 'eListenEintrag' dar. Jeder Eintrag besteht aus einem im Moment uninteressanten 'inhalt' und einer zeigerwertigen Komponente 'nachfolger' vom Typ 'eListe'. Da 'eListe' an den Typ 'eListenEintrag' gebunden ist, kann 'nachfolger' nur auf dynamische Variablen des Typs 'eListenEintrag' verweisen.

Bei der Vereinbarung von Pointertypen gibt es die einzige Ausnahme von der im Abschnitt 5.4 formulierten Regel, dass ein Bezeichner vor seiner Benutzung in einer anderen Vereinbarung vereinbart sein muss:

> **Die Vereinbarung eines Pointertyps P, in der ein Typbezeichner TNAME verwendet wird, kann textlich vor der Vereinbarung von TNAME stehen, wenn TNAME noch in demselben Block vereinbart wird.**

Schon das Beispiel 'eListe' zeigt, dass diese Ausnahmeregelung notwendig ist. Wie sollte man sonst innerhalb des Typs 'eListenEintrag' Zeiger auf Records desselben Typs vereinbaren?

Werte aus Pointertypen können mittels der Operatoren '=', '<>' und '#' auf Gleichheit bzw. Ungleichheit getestet werden. Darüber hinaus sind nur noch Wertzuweisungen und die Verwendung als Parameter bzw. Funktionsresultat zulässig. Die mit Pointertypen in Verbindung stehenden Standardprozeduren werden an späterer Stelle dieses Abschnitts erläutert.
Der Standardbezeichner 'NIL' beschreibt einen speziellen Zeiger, der zu jedem Pointertyp gehört und auf keine Variable verweist. Er wird verwendet, um nachprüfbar fixieren zu können, dass eine Zeigervariable nirgendwohin zeigt. Im Bild 12.1 ist 'NIL' durch das elektrotechnische Erdezeichen symbolisiert und dient zur Anzeige des Kettenendes.
Die Notation des Zugriffs auf dynamische Variablen ordnet sich in die Syntaxregeln 'bezugnahme' und 'selektor' aus Abschnitt 9.2.1 ein.

```
$    pointer_zugriff = "^".
   Die 'bezugnahme' vor "^" muss von einem Pointertyp sein.
```

Das "Dach" kann man als Verweispfeil interpretieren. Wenn 'xyz' ein Zeiger auf eine dynamische Variable ist, dann bezeichnet 'xyz^' diese dynamische Variable. Die folgende kleine Tabelle bezieht sich auf die im Bild 12.1 gezeichnete Situation und die zugehörigen Vereinbarungen:

Anfang	statische Variable vom Typ 'eListe', Zeiger auf das linke grosse Kästchen
Anfang^	dynamische Variable vom Typ 'eListenEintrag', das linke grosse Kästchen
Anfang^.inhalt	Inhalt des linken grossen Kästchens
Anfang^.nachfolger	Zeiger auf das zweite grosse Kästchen
Anfang^.nachfolger^	das zweite grosse Kästchen.

Es ist gewiss nicht sehr sinnvoll, für den Zugriff auf den Inhalt des letzten grossen Kästchens

```
   Anfang^.nachfolger^.nachfolger^.nachfolger^.inhalt
```

zu notieren. Was mache ich, wenn die Liste 7 Elemente enthält? Was passiert, wenn nur zwei Einträge da sind? Ich habe mir Bezugnahmen mit mehr als einem Pointerzugriff abgewöhnt. Wenn man tiefer in eine verzeigerte Datenstruktur hinein muss, dann nehme man sich

eine Hilfsvariable vom entsprechenden Pointertyp und "schiebe" sie mit Hilfe geeigneter Wertzuweisungen von einer dynamischen Variablen zur anderen, bis man an der gewünschten Stelle ist. Die folgenden zwei Prozeduren demonstrieren diese Verfahrensweise.
'Zaehlen' stellt fest, wie viele Listeneinträge vorhanden sind.

```
PROCEDURE Zaehlen(VAR anzahl: INTEGER);
VAR  h: eListe;
BEGIN
  h:=Anfang; anzahl:=0;
  WHILE h#NIL DO INC(anzahl); h:=h^.nachfolger END;
END Zaehlen;
```

Wenn Ihnen Pointertypen und dynamische Datenstrukturen neu sind, dann spielen Sie bitte an Hand von Bild 12.1 einen Lauf dieser Prozedur durch! Ich empfehle die Verwendung eines Ihrer Finger als Hilfsvariable 'h'.
Die Funktionsprozedur 'Suche' sucht einen 'eListenEintrag' mit vorgegebenem Inhalt (Parameter 'wert'). Sie liefert als Resultat entweder einen Zeiger auf den ersten Listeneintrag mit dem gewünschten 'inhalt' oder 'NIL'.

```
PROCEDURE Suche(wert: INTEGER): eListe;
VAR  h: eListe;
BEGIN
  h:=Anfang;
  WHILE (h#NIL) AND (h^.inhalt#wert) DO h:=h^.nachfolger END;
  RETURN h;
END Suche;
```

Die folgende Bemerkung ist inhaltlich sofort einsehbar. Sie betrifft den im Zusammenhang mit Pointertypen am häufigsten auftretenden Programmierfehler.

Pointerzugriffe 'p^' sind nur dann möglich, wenn der Zeiger 'p' auch wirklich auf eine dynamische Variable zeigt. Für 'p=NIL' führt die Abarbeitung von 'p^' zum Programmabbruch.

In den Prozeduren 'Zaehlen' und 'Suche' sind die Bezugnahmen 'h^.nachfolger' durch die Bedingung 'h#NIL' geschützt. Bei 'Suche' wird der zweite Teil der WHILE-Bedingung im Falle 'h=NIL' nicht ermittelt (vgl. Abschn. 6.5). Es kann also nichts schiefgehen.
Was geschieht, wenn man über eine Pointervariable zugreift, der nie ein Wert zugewiesen wurde, das ist leider ungewiss. Wenn man Glück

hat, dann gibt es einen Programmabbruch. Es kann aber auch sein, dass die Abarbeitung mit irgendwelchen Werten fortschreitet und üble, nicht zurückzuverfolgende Spätschäden auftreten. Man sollte sich deshalb in seinen Programmen nie darauf verlassen, dass einem Zeiger "später" ein Wert zugewiesen wird, sondern vorsichtshalber zunächst 'NIL' eintragen.
Aus der Syntax von MODULA-2 ergibt sich, dass an einen Funktionsaufruf kein Pointerzugriff "angehängt" werden kann. Für Zugriffe über das Resultat zeigerwertiger Funktionsprozeduren benötigt man eine Hilfsvariable vom entsprechenden Pointertyp.
Die Notation 'Suche(17)^' wird von meinem Compiler mit der recht unscharfen Fehlermeldung "error in expression" zurückgewiesen. Wenn 'h' eine passende Variable ist, dann leistet 'h:=Suche(17); h^' das Verlangte.

Dynamische Variablen werden in MODULA-2 durch Rufe der Standardprozedur 'NEW' erzeugt. Sie können durch Rufe der Standardprozedur 'DISPOSE' wieder vernichtet werden. Hinter 'NEW' verbirgt sich die Zuteilung von Speicherplatz und hinter 'DISPOSE' eine Speicherplatzfreigabe.
Gegeben sei eine Pointervariable 'VAR p: POINTER TO T'. Der Prozedurruf 'NEW(p)' bewirkt zweierlei:

1. Es wird Speicherplatz für ein Objekt des Typs 'T', an den der Pointer 'p' gebunden ist, reserviert.
2. Der als Parameter übergebenen Variablen 'p' wird ein Zeigerwert zugewiesen, der auf dieses neue Objekt verweist.

Die Wirkung des Aufrufs 'NEW(p)' ist unabhängig von dem Wert, den der aktuelle Parameter 'p' vorher hat. Der Parameter ist ein Referenzparamter. Deshalb sind als aktuelle Parameter nicht nur pointerwertige Variablen, sondern auch pointerwertige Record- und Arraykomponenten sowie pointerwertige Parameter möglich.
Was geschieht, wenn der verfügbare Speicherplatz nicht mehr für ein Objekt des verlangten Typs ausreicht? Die Antwort auf diese Frage ist implementationsabhängig. Zwei Standardvarianten sind üblich: Programmabbruch oder Zuweisung des Wertes 'NIL' an den übergebenen Parameter. Ich diskutiere dieses Problem in Zusammenhang mit dem Bibliotheksbustein 'Storage' ausführlicher.
Für Pointertypen, die an Records mit Variantenteilen gebunden sind,

gibt es die Möglichkeit zur Angabe weiterer Parameter beim Aufruf von 'NEW'. Man kann problemlos darauf verzichten.
Als Beispiele für die Erzeugung dynamischer Variablen folgen vier Prozeduren zum Erweitern der einfach verketteten Liste. Ich beginne mit dem Einfachsten: Einfügen eines neuen Elements am Listenanfang.

```
PROCEDURE Vorn(wert: INTEGER);
VAR  h: eListe;
BEGIN
  NEW(h); WITH h^ DO inhalt:=wert; nachfolger:=Anfang END;
  Anfang:=h;
END Vorn;
```

Diese Prozedur funktioniert auch dann, wenn die Liste leer war, also noch gar kein Element enthielt. Für einen ordnungsgemässen Aufbau entsprechend Bild 12.1 muss gefordert werden, dass 'Anfang' den Anfangswert 'NIL' hat. Ich setze das auch in den folgenden Prozeduren voraus. Wie fügt man am Listenende an?

```
PROCEDURE Hinten(wert: INTEGER);
VAR  h: eListe;
BEGIN
  IF Anfang#NIL THEN
    h:=Anfang;
    WHILE h^.nachfolger#NIL DO h:=h^.nachfolger END;
    NEW(h^.nachfolger); h:=h^.nachfolger;
  ELSE
    NEW(Anfang); h:=Anfang;
  END;
  WITH h^ DO inhalt:=wert; nachfolger:=NIL END;
END Hinten;
```

Der erste Aufruf von 'NEW' trägt den Zeiger auf das neue Objekt sofort in das 'nachfolger'-Datenfeld des bisher letzten Listenelements ein. Gewiss fallen Ihnen nach kurzem Nachdenken Modifikationen dieser Prozedur ein. Probieren Sie sie aus!
Beim Einfügen hinter oder vor einem bestimmten Element leistet die Funktionsprozedur 'Suche' gute Hilfsdienste. Der Parameter 'w' gibt den Inhalt desjenigen Listenelements an, hinter bzw. vor dem eingefügt werden soll. Wenn sich kein Eintrag mit dem fraglichen Inhalt findet, dann machen die Prozeduren gar nichts. Abhängig von der konkreten Verwendung einer einfach verketteten Liste wären auch

andere Lösungen möglich und sinnvoll. Die folgende Prozedur 'Nach' legt einen neuen 'eListenEintrag' an und "kettet" ihn hinter dem mittels 'Suche' ermittelten Listenelement ein.

```
PROCEDURE Nach(w,wert: INTEGER);
VAR  h,q: eListe;
BEGIN
  h:=Suche(w); IF h=NIL THEN RETURN END;
  NEW(q);
  WITH q^ DO inhalt:=wert; nachfolger:=h^.nachfolger END;
  h^.nachfolger:=q;
END Nach;
```

Für das Einfügen vor einem gegebenen Listenelement gibt es eine sehr elegante Lösung. Man sucht den fraglichen Eintrag E und fügt hinter ihm einen neuen Eintrag N in die Liste ein. Jetzt wird N mit dem Inhalt von E belegt, und danach wird der neue Inhalt (Parameter 'wert') in E eingeschrieben. In der folgenden Prozedur haben die Pointerzugriffe 'e^' und 'n^' die Bedeutung von E bzw. N.

```
PROCEDURE Vor(w,wert: INTEGER);
VAR e,n: eListe;
BEGIN
  e:=Suche(w); IF e=NIL THEN RETURN END;
  NEW(n); n^:=e^; e^.nachfolger:=n; e^.inhalt:=wert;
END Vor;
```

Die Anweisung 'n^.nachfolger:=e^.nachfolger' (d. h. eine Hälfte des Einkettens) wurde gleich mit dem Umspeichern des Inhalts ('n^.inhalt:=e^.inhalt') zu 'n^:=e^' zusammengelegt.

Gegeben sei eine zeigerwertige Variable 'VAR p: POINTER TO T'. Der Prozedurruf 'DISPOSE(p)' meldet den von der dynamischen Variablen 'p^' belegten Speicherplatz frei. 'DISPOSE' ist der einzige Weg zu einem Recycling des Speicherplatzes. Wenn eine dynamische Variable erzeugt wurde, dann existiert sie bis zu ihrer expliziten Freigabe. Das trifft auch dann zu, wenn gar kein Pointer mehr auf dieses Objekt verweist.

Wichtige Bemerkungen zur Arbeitsweise von 'DISPOSE(p)':

1. Im allgemeinen wird der von 'p^' belegte Speicherplatz nicht gelöscht, sondern es wird nur intern vermerkt, dass er wieder für NEW-Operationen zur Verfügung steht.

2. Obwohl der Parameter ein Referenzparameter ist, steht nicht fest, welchen Wert 'p' nach der Speicherfreigabe hat.
3. Was bei "ungültigen" Werten von 'p' geschieht, das ist implementationsabhängig.

Aus dem Gesagten folgt, dass der Programmierer vor dem Aufruf von 'DISPOSE' die betreffende dynamische Variable sorgfältig aus seiner Datenstruktur ausgliedern muss.
Als Beispiel gebe ich zwei Prozeduren zum Streichen von Einträgen in einer einfach verketteten Liste an. Die Freigabe des ersten Listenelements macht keine Schwierigkeiten. Wenn die leere Liste vorliegt, dann ist nichts zu tun.

```
PROCEDURE KopfAb;
VAR  h: eListe;
BEGIN
  IF Anfang=NIL THEN RETURN END;
  h:=Anfang; Anfang:=Anfang^.nachfolger; DISPOSE(h);
END KopfAb;
```

Beim Streichen eines Listenelements mit vorgegebenem Inhalt ist die Prozedur 'Suche' wenig hilfreich, da ich auch am Vorgänger des zu streichenden Eintrags etwas ändern muss.

```
PROCEDURE Streichen(w: INTEGER);
VAR h1,h2: eListe;
BEGIN  h1:=Anfang;
  WHILE (h1#NIL) AND (h1^.inhalt#w) DO
    h2:=h1; h1:=h1^.nachfolger;
  END;
  IF h1=NIL THEN RETURN END;
  IF h1=Anfang THEN Anfang:=h1^.nachfolger;
  ELSE h2^.nachfolger:=h1^.nachfolger END;
  DISPOSE(h1);
END Streichen;
```

Wenn wir aus den bisher angegebenen Typ-, Variablen- und Prozedurvereinbarungen einen Programmodul machen und diesen übersetzen, dann erleben wir eine böse Überraschung. Mein Compiler liefert bei jedem Ruf von 'NEW' bzw. 'DISPOSE' die Fehlermeldung

```
124: no procedure found for substitution.
```

Wie kommt das? Die MODULA-2-Compiler verfolgen bez. der Standardprozeduren 'NEW' und 'DISPOSE' eine interessante Strategie.

Sei 'p' eine an den Typ 'T' gebundene Pointergrösse. Die Notation 'TSIZE(T)' drückt den Speicherplatzbedarf für ein Objekt des Typs 'T' aus.

1. Für die Standardprozedurrufe 'NEW(p)' und 'DISPOSE(p)' werden intern "gewöhnliche" Prozeduraufrufe 'ALLOCATE(p,TSIZE(T))' und 'DEALLOCATE(p,TSIZE(T))' eingesetzt.
2. Der Programmierer hat dafür zu sorgen, dass 'ALLOCATE' bzw. 'DEALLOCATE' gültige Bezeichner an den Programmstellen sind, wo die Namen 'NEW' bzw. 'DISPOSE' verwendet werden. Das heisst, er muss diese Prozeduren entweder selbst vereinbaren oder von irgendwoher importieren.

Weshalb wurde ein solcher Kopfstand erfunden? Er bringt eine grosse Flexibilität! Jeder Programmierer hat die Möglichkeit, sich eine seinen speziellen Bedürfnissen angepasste private Freispeicherverwaltung zu schreiben und sie mit Hilfe der Standardprozeduren 'NEW' bzw. 'DISPOSE' zu benutzen.

Bei unserem Beispiel wurden weder 'ALLOCATE' noch 'DEALLOCATE' vereinbart oder importiert. Folglich hat der Compiler Schwierigkeiten bei der Substitution und teilt das als Quelltextfehler mit. Zur Lösung unseres Problems können wir auf eine vorgefertigte Speicherverwaltung zurückgreifen. Unter den mit jedem MODULA-2-System gelieferten Bibliotheksbausteinen gibt es immer einen mit dem Namen 'Storage', der die gefragten Prozeduren exportiert. Durch Einfügen der Zeile

```
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
```

schliessen wir die Standardprozeduren 'NEW' und 'DISPOSE' an die im Baustein 'Storage' programmierte Speicherverwaltung an.

Im allgemeinen ist die von 'Storage' angebotene Speicherzuteilung und -freigabe völlig ausreichend. Bedarf nach Speziallösungen entsteht äusserst selten. Das rechtfertigt den folgenden Merksatz:

> **Jeder Programm- oder Implementationsmodul, der dynamische Variablen anfordert oder freigibt, muss die Importklausel**
>
> ```
> FROM Storage IMPORT ALLOCATE, DEALLOCATE;
> ```
>
> **enthalten.**

Die mit den unterschiedlichen MODULA-2-Systemen angebotenen Speicherverwaltungsbausteine 'Storage' unterscheiden sich leider bez. der Reaktion auf Anforderungen, die wegen Speicherplatzmangels nicht befriedigt werden können. Wie kann man vom Programm aus

feststellen, dass der Speicherplatz ausgeschöpft ist? Zwei Varianten sind verbreitet.
Bei der einen Variante (s. [Logi85], [Robo87b]) führt jede nicht mehr zu befriedigende Anforderung zum Programmabbruch. Mittels der zusätzlich von 'Storage' exportierten

```
PROCEDURE Available(size: CARDINAL): BOOLEAN;
```

kann erfragt werden, ob noch Speicherplatz der Groesse 'size' vorhanden ist. Für 'VAR p: POINTER TO T;' führt das zu dem folgenden Programmiermuster:

```
IF Available(TSIZE(T)) THEN NEW(p) ELSE ... END;
```

Die andere Variante ([Wirt80]) stellt eine Prozedur bereit, die das Verhalten von 'ALLOCATE' beeinflusst.

```
PROCEDURE SetMode(m: CARDINAL);
```

Normalerweise führt eine nicht zu befriedigende Anforderung zum Programmabbruch. Durch den Aufruf 'SetMode(2)' wird veranlasst, dass 'ALLOCATE' (und damit 'NEW') im Fall nicht mehr ausreichenden Speicherplatzes fortan den Pointerwert 'NIL' liefern. Das Programmiermuster nimmt damit etwa folgende Gestalt an:

```
SetMode(2); ...
NEW(p); IF p=NIL THEN ... ELSE ... END;
```

Ein Aufruf von 'SetMode' mit dem Parameterwert 1 führt wieder in den Normalzustand (Abbruch) zurück.
Die Dokumentation Ihres MODULA-2-Systems enthält gewiss einen breit kommentierten Definitionsmodul 'Storage', dem Sie die konkreten Angaben entnehmen können.

12.2. Binäre Bäume

"Speichern und Wiederfinden" ist ähnlich dem "Sortieren" eine Standardaufgabe bei der Arbeit mit Computern. Ich setze voraus, dass die zu behandelnde Datenmenge aus Datensätzen fester Struktur besteht. Die Datensätze sollen sich aus einem Kennzeichnungsfeld (Schlüssel) und einem eigentlichen Inhalt zusammensetzen, und für den Typ des Schlüssels soll eine Ordnungsrelation erklärt sein. Wiederfinden heisst: Herausfischen eines Datensatzes an Hand eines vorgegebenen Schlüssels.
Die Aufgabe "Speichern und Wiederfinden" kann unter den genannten

Bedingungen günstig mittels der Datenstruktur "binärer Baum" gelöst werden. Ein binärer Baum besteht aus Knoten, in denen Schlüssel und Inhalt sowie Zeiger auf zwei Nachfolgeknoten gespeichert sind. Auf jeden Knoten verweist genau ein Zeiger.

```
TYPE
    BinBaum = POINTER TO Knoten;
    Knoten = RECORD
               schluessel: INTEGER;
               inhalt: INTEGER;
               links,rechts: BinBaum;
             END;
VAR
  Wurzel: BinBaum;
```

Der Einfachheit halber habe ich den Schlüsseltyp 'INTEGER' gewählt. Es wären auch andere Schlüssel möglich, z. B. Zeichenketten mit lexikografischer Ordnung oder Uhrzeiten. Der Typ von 'inhalt' ist zunächst uninteressant. Ich habe 'INTEGER' genommen, weil das zu einem später folgenden Beispielprogramm passt.

Der Baum sei so aufgebaut, dass für jeden Knoten K folgende Aussagen gelten:

1. Die Schlüsselwerte aller Knoten, die am 'links'-Zeiger von K hängen, sind kleiner als der Schlüssel von K.
2. Die Schlüsselwerte aller Knoten, die am 'rechts'-Zeiger von K hängen, sind grösser als der Schlüssel von K.

Im Bild 12.2 ist ein derartiger binärer Baum dargestellt. Das Datenfeld 'inhalt' wurde weggelassen.

Das Suchen eines Knotens mit vorgegebenem Schlüssel kann leicht als rekursive Prozedur formuliert werden. Ich setze voraus, dass der gesuchte Schlüsselwert 's' in dem Baum, auf dessen ersten Knoten 'b' verweist, wirklich vorhanden ist.

```
PROCEDURE Suche(b: BinBaum; s: INTEGER; VAR i: INTEGER);
BEGIN
  WITH b^ DO
    IF schluessel=s THEN (* Gefunden! *) i:=inhalt;
    ELSIF s<schluessel THEN Suche(links,s,i);
    ELSE (* schluessel<s *) Suche(rechts,s,i);
  END;
END Suche;
```

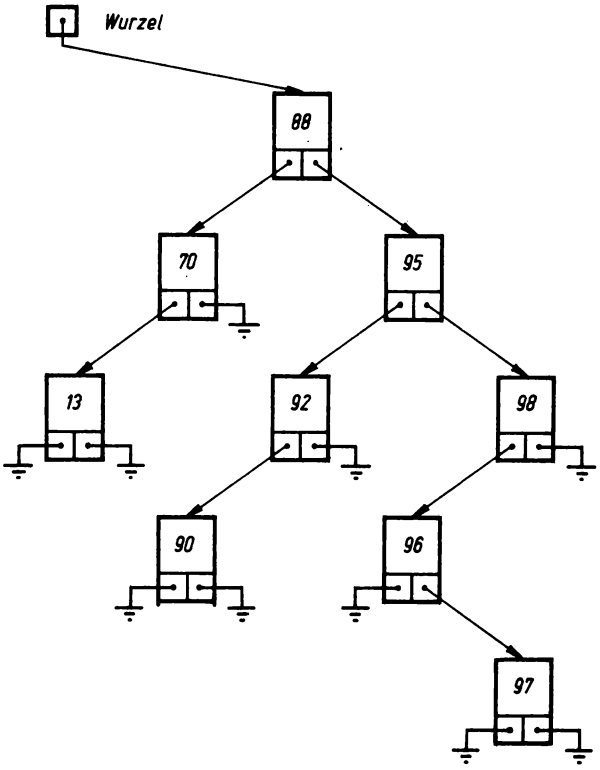


Bild 12.2. Ein binärer Baum mit ganzzahligen Schlüsselwerten

Die Idee der Prozedur 'Suche' ist sehr einfach: Wenn der gesuchte Schlüssel 's' nicht in 'b^' selbst steht, dann muss er entweder unter den linken oder unter den rechten Nachfolgern gesucht werden. Die Frage, ob links oder rechts, kann durch den Vergleich von 'b^.schluessel' mit dem Parameter 's' entschieden werden. Und zum Suchen im jeweiligen Teilbaum benutzt man am besten wieder die Prozedur 'Suche'. Bitte spielen Sie den Aufruf 'Suche(Wurzel,96,x)' auf Bild 12.2 durch!
Wenn ein Knoten keinen linken und/oder rechten Nachfolger hat, dann wird das durch den Wert 'NIL' in den entsprechenden Datenfeldern angezeigt. Bitte fassen Sie die 'NIL'-Werte als "leere Bäume" auf! Dieses Denkmodell bewährt sich. Beim Entwerfen von Prozeduren für die Arbeit mit Binärbäumen muss man sich immer fragen: Was ist mit

dem leeren Baum zu tun, und wie kann ich die gewünschte Aktion aus den Bearbeitungen des linken und des rechten Teilbaums sowie der Behandlung des aktuellen Wurzelknotens zusammensetzen? Wenn wir dieses Denkschema auf die Aufgabe "Ausschreiben aller in einem Baum auftretenden Schlüssel, aufsteigend sortiert" anwenden, dann ergibt sich folgendes:

- Mit dem leeren Baum ist gar nichts zu tun.
- Wenn ich zuerst den linken Teilbaum aufsteigend sortiert ausschreibe und dann den Schlüsselwert des Wurzelknotens anfüge und zum Schluss noch den rechten Teilbaum aufsteigend sortiert ausgebe, dann ist alles erledigt.

Hieraus ergibt sich sofort die folgende Prozedur, in der eine Importklausel 'IMPORT InOut;' vorausgesetzt wird:

```
PROCEDURE Schreibe(b: BinBaum);
BEGIN
  IF b=NIL THEN RETURN END;
  Schreibe(b^.links);
  InOut.WriteInt(b^.schluessel); InOut.WriteLn;
  Schreibe(b^.rechts);
END Schreibe;
```

Auch diese Prozedur sollten Sie an Hand von Bild 12.2 durchspielen! Derartige Prozeduren sind sehr sensibel bez. der richtigen Reihenfolge zwischen den Teilbaumbearbeitungen und der Wurzelbehandlung. Wenn man in 'Schreibe' die beiden Aufrufe von 'Schreibe' miteinander vertauscht, dann werden die Schlüssel absteigend sortiert ausgeschrieben.

Programm 12.1 fasst die Operationen "Aufbauen eines binären Baums", "Suchen in diesem Baum" und "sortiertes Ausschreiben" in einem einfachen Beispiel zusammen. Das Programm liest eine Folge nichtnegativer Zahlen ein. Die erste negative Zahl wird als Endekennzeichen aufgefasst. Das Programm registriert, wie oft jede Zahl vorkommt, und gibt abschliessend die erfassten Zahlen und ihre Häufigkeiten aus.

Die Prozedur 'Verbuchen' (Zeilen 20 bis 3) folgt dem Schema von 'Suche'. Sie behandelt aber auch den Fall, dass der gesuchte Schlüssel nicht vorhanden ist. Der Parameter 'b' muss jetzt ein Referenzparameter sein, damit der von 'NEW' in Zeile 27 gelieferte Pointer auch wirklich beim aktuellen Parameter "ankommt".

```
 1 MODULE BinaerBaum;
 2 (*-------------------------------------------------------------
 3   Demonstrationsbeispiel zum "Suchen und Wiederfinden" mit
 4   binaeren Baeumen:
 5       Ermitteln der Haeufigkeiten von Eingabezahlen
 6 -------------------------------------------------------------*)
 7 FROM Storage IMPORT ALLOCATE;
 8 FROM InOut IMPORT
 9   ReadInt, Done, WriteString, WriteInt, WriteLn;
10 (*-------------------------------------------------------------*)
11 TYPE
12   BinBaum = POINTER TO Knoten;
13   Knoten = RECORD
14              schluessel,inhalt: INTEGER;
15              links,rechts: BinBaum;
16            END;
17 VAR
18   Wurzel: BinBaum;
19
20 PROCEDURE Verbuchen(VAR b: BinBaum; zahl: INTEGER);
21 (* Verbuchen eines Auftretens von 'zahl' im Baum 'b'
22    Das heisst: 'INC(inhalt)' im Knoten mit 'schluessel=zahl'.
23    Wenn 'zahl' noch nicht als Schluessel auftritt, dann muss
24    ein neuer Knoten mit 'inhalt=1' angelegt werden.          *)
25 BEGIN
26   IF b=NIL THEN (* 'zahl' noch nicht gespeichert, Neueintrag *)
27     NEW(b);
28     WITH b^ DO
29       schluessel:=zahl; inhalt:=1; links:=NIL; rechts:=NIL;
30     END;
31     RETURN;
32   END;
33   WITH b^ DO
34     IF schluessel=zahl THEN INC(inhalt);
35     ELSIF zahl<schluessel THEN Verbuchen(links,zahl);
36     ELSE Verbuchen(rechts,zahl) END;
37   END;
38 END Verbuchen;
```

```
 39 PROCEDURE Schreiben(b: BinBaum);
 40 (* 'schluessel' und 'inhalt' entspr. 'schluessel' aufsteigend
 41    sortiert ausschreiben.                                    *)
 42 BEGIN
 43   IF b=NIL THEN RETURN END;
 44   WITH b^ DO
 45     Schreiben(links);
 46     WriteInt(schluessel,6); WriteInt(inhalt,4); WriteLn;
 47     Schreiben(rechts);
 48   END;
 49 END Schreiben;
 50 (*-----------------------------------------------------------*)
 51 VAR
 52   n: INTEGER;
 53 (*-----------------------------------------------------------*)
 54 BEGIN
 55   WriteString("Bitte Zahlen eingeben!"); WriteLn;
 56   WriteString("Ein negativer Wert beendet die Eingabe.");
 57   WriteLn; Wurzel:=NIL;
 58   LOOP
 59     REPEAT ReadInt(n); WriteLn UNTIL Done;
 60     IF n<0 THEN EXIT END;
 61     Verbuchen(Wurzel,n);
 62   END;
 63   Schreiben(Wurzel);
 64 END BinaerBaum.
```

Programm 12.1. Demonstrationsbeispiel für binäre Bäume

Im Anweisungsteil des Programmoduls wird nach einigen freundlichen Ausschriften die Pointervariable 'Wurzel' initialisiert (Zeile 57). Wenn eine akzeptable Zahl eingelesen wurde, dann nehme ich sie in die Buchführung auf (Zeile 61).
Die interne Struktur des von Programm 12.1 aufgebauten Binärbaums hängt wesentlich von der Reihenfolge des ersten Auftretens der Zahlen ab. Wenn sie geordnet eingegeben werden, dann degeneriert der Baum de facto zu einer einfach verketten Liste. Welche Eingabefolgen führen zu der Struktur von Bild 12.2?

13. Vervollständigung der Sprachdarstellung

Die in den Abschnitten 5 bis 12 enthaltene Syntax von MODULA-2 ist nahezu vollständig. Nur für das in der Regel 'vereinbarung' von Abschnitt 9 auftretende Hilfssymbol 'modulvereinbarung' fehlt noch eine erklärende Syntaxregel. Abschnitt 13.1 schliesst diese Lücke. Lokale Module sind eine gutgemeinte Erfindung, die sich aber in der Programmierpraxis nicht bewährt hat [Wirt87].
Der Standardbaustein 'SYSTEM' stellt einige Ausdrucksmittel für die maschinenunabhängige maschinennahe Programmierung bereit. Im Abschnitt 13.2 wird diese scheinbar paradoxe Formulierung aufgelöst. Für die von 'SYSTEM' exportierten Bezeichner werden einige der im Abschnitt 9.5 formulierten Regeln bez. Typgleichheit und -verträglichkeit sowie Zuweisungsverträglichkeit arg aufgeweicht. Aus diesem Grunde kann 'SYSTEM' nicht als gewöhnlicher separater Baustein existieren. Er ist fest in den MODULA-2-Compiler integriert. Ich bezeichne ihn daher als Pseudobaustein.
Niklaus Wirth hat in seiner sehr eleganten MODULA-Weiterentwicklung OBERON [Wirt87] sowohl die lokalen Module als auch die Hintertür 'SYSTEM' ersatzlos gestrichen.

13.1. Lokale Module

MODULA-2 bietet die Möglichkeit, in jedem 'block' (s. Abschn. 5.1) neben Konstanten, Typen, Variablen und Prozeduren auch noch lokale oder innere Module zu vereinbaren.

```
$    modulvereinbarung = MODULE bezeichner [prioritaet] ";"
$                        {import ";"} [export] block bezeichner .
   Hinter MODULE und nach 'block' muss derselbe 'bezeichner'
   notiert werden (Modulname)
$    prioritaet = "[" konstantenausdruck "]"
$    export = EXPORT [QUALIFIED] bezeichnerliste .
```

Ein innerer Modul ist eine abgeschlossene Kapsel, die über einige wohldefinierte Fäden ('export' und 'import') mit ihrer Umgebung

verbunden ist. Im Modulblock sind nur die lokalen Bezeichner des Modulblocks und die importierten Bezeichner gültig. Importe legen also fest, welche Objekte der Aussenwelt in der Kapsel sichtbar sind. Die Aussenwelt kann von der Kapsel nur diejenigen Dinge wahrnehmen (sprich: benutzen), die exportiert werden. Alles andere bleibt ein verkapseltes Geheimnis.
Wenn in der Exportklausel das Schlüsselwort QUALIFIED steht, dann sind Bezugnahmen auf die exportierten Dinge nur mittels 'qname' (bestehend aus Modulname, Punkt und exportiertem Bezeichner) möglich. Es sei denn, man sitzt in einer anderen Kapsel und importiert die fraglichen Bezeichner mit Hilfe einer FROM-IMPORT-Klausel.
Wegen der Möglichkeit, lokale Module zu verschachteln, bekommt die Syntaxregel 'q_name' aus Abschnitt 5.1 die folgende endgültige Gestalt:

$ q_name = {modulname "."} bezeichner .
Der erste 'modulname' kann ein Bausteinname sein; alle weiteren müssen Namen lokaler Module sein.

Ein innerer Modul hat dieselbe Lebensdauer wie der Block, zu dem er lokal ist (bez. "Lebensdauer" vgl. Abschn. 5.4). Der Anweisungsteil eines Modulblocks wird unmittelbar vor der Abarbeitung des Anweisungsteils des den Modul enthaltenden Blocks ausgeführt.
Die Angabe einer 'prioritaet' ist nur in einigen MODULA-2-Systemen erlaubt und spielt dort bei sehr speziellen Anwendungen des Coroutinenkonzepts eine Rolle.

Das, was ich hier über lokale Module aufgeschrieben habe, reicht aus, um Programme mit lokalen Modulen verstehen zu können. Ich empfehle Ihnen, auf lokale Module zu verzichten. In den Programmen, die ich gesehen habe, wurden lokale Module entweder nur benutzt, um auch dieses Sprachelement einmal ausprobiert zu haben, oder sie konnten vorteilhaft in Bausteine übergeführt werden.

13.2. Der Pseudobaustein 'SYSTEM'

Programme mit Maschinenabhängigkeiten sind nicht portabel. Auf einem anderen Computer kann vieles ganz anders sein. Wie kann man feststellen, ob ein MODULA-2-Programm maschinenabhängig ist?
Ein gewisser Teil der portabilitätsfeindlichen Sprachelemente wurde im Pseudobaustein 'SYSTEM' versteckt. Wenn ein Implementations- oder Programmodul von 'SYSTEM' importiert, dann ist er dringend der Maschinenabhängigkeit verdächtig. Darüber hinaus erlaubt MODULA-2 zwei weitere maschinenbezogene Ausdrucksmittel: Adressangaben bei Variablenvereinbarungen und "Typtransfer" für Ausdrücke.
Wenn in einer Variablenvereinbarung hinter dem Variablennamen in eckigen Klammern ein CARDINAL-wertiger Konstantenausdruck angegeben wird, dann entzieht der Compiler diese Variable der normalen MODULA-2-Speicherzuteilung. Der Wert des Konstantenausdrucks wird als absolute Maschinenadresse für die Variable angesehen. Aus einem Programm heraus kann auf spezielle, physisch festliegende Speicherplätze Bezug genommen werden. Aber wehe, wenn Sie mit einem derartigen Programm auf einen anderen Rechnertyp umsteigen! Oft ist sogar schon ein Wechsel der Betriebssystemversion auf demselben Computer tödlich.
Die syntaktischen Eigenschaften der im Pseudobaustein 'SYSTEM' verfügbaren Sprachelemente werden näherungsweise durch den "Definitionsmodul" auf der folgenden Seite beschrieben. Dargestellt ist der in der Sprachdefinition fixierte Minimalbestand. Jedes konkrete MODULA-2-System fügt i. allg. weitere spezifische Dinge hinzu.
Der Typ 'WORD' beschreibt ein "Speicherwort". Für Objekte dieses Typs sind keine Operationen erklärt, aber man kann Wertzuweisungen ausführen. Interessant und sinnvoll sind formale Parameter vom Typ

    WORD oder ARRAY OF WORD.

Auf der Position eines 'WORD'-Parameters sind in Prozedur- und Funktionsaufrufen aktuelle Parameter zulässig, deren Speicherplatzbedarf gerade ein Speicherwort beträgt und deren Typ kein Array- oder Recordtyp ist. Für 'ARRAY-OF-WORD'-Parameter können beliebige aktuelle Parameter vermittelt werden, deren Speicherplatzbedarf ein ganzzahliges Vielfaches des Speicherplatzbedarfs von 'WORD' ist.

```
DEFINITION MODULE SYSTEM;
(*-------------------------------------------------------------
     Pseudobaustein, der maschinenabhängige Dinge
     bereitstellt
-------------------------------------------------------------*)
EXPORT QUALIFIED
     WORD, ADDRESS,
     ADR, SIZE, TSIZE, NEWPROCESS, TRANSFER;
(*-------------------------------------------------------------*)
TYPE
  WORD;
  ADDRESS = POINTER TO WORD;

PROCEDURE ADR(VAR x: ...): ADDRESS;

PROCEDURE SIZE(VAR x: ...): CARDINAL;
PROCEDURE TSIZE(typname): CARDINAL;

PROCEDURE NEWPROCESS(p: PROC; b: ADDRESS; l: CARDINAL;
                     VAR co: ADDRESS                 );
PROCEDURE TRANSFER(co1,co2: ADDRESS);
(*-------------------------------------------------------------*)
END SYSTEM.
```

Der typische Einsatzfall für 'WORD' sind Prozeduren zum Umspeichern von Daten, die für Objekte beliebiger Typen funktionieren.

```
PROCEDURE Vertauschen(VAR a,b: ARRAY OF WORD);
VAR  i: INTEGER;  w: WORD;
BEGIN
  FOR i:=0 TO INTEGER(HIGH(a)) DO
    w:=a[i]; a[i]:=b[i]; b[i]:=w;
  END;
END Vertauschen;
```

Der zweite von 'SYSTEM' exportierte Typ - 'ADDRESS' - ist mit allen Pointertypen verträglich. Auf 'ADDRESS'-Werte können alle für den

Typ 'CARDINAL' erklärten arithmetischen Operationen, Standardprozeduren und Standardfunktionen angewendet werden. Die genaue Wirkung solcher Adressrechnungen ist nicht nur maschinenspezifisch, sie hängt auch vom Betriebssystem und dem jeweiligen MODULA-2-System ab.

Die Funktion 'ADR' liefert die Adresse der als aktueller Parameter notierten 'bezugnahme'. Kurzsichtige Programmierer benutzen diese Hintertür, um entgegen den Prinzipien des Pointerkonzepts der Sprache MODULA-2 auch mit Zeigern auf vereinbarte Variablen und auf Komponenten von Variablen zu hantieren.

Die Funktionen 'SIZE' und 'TSIZE' liefern den Speicherplatzbedarf der als Parameter notierten 'bezugnahme' bzw. des als Parameter notierten Typs. Man weiss nicht, in welcher Einheit hier gezählt wird. Das heisst, 'TSIZE(WORD)=1' muss nicht unbedingt gelten. Im allgemeinen gilt 'TSIZE(INTEGER)=TSIZE(CARDINAL)=TSIZE(WORD)'.

Die bisher erläuterten Sprachelemente aus 'SYSTEM', insbesondere der Typ 'WORD' sowie die Funktionen 'SIZE' und 'TSIZE', erwecken den Eindruck, als seien maschinenunabhängige Ausdrucksmittel für maschinenabhängige Dinge gefunden worden: Der Compiler nimmt beim Übersetzen die Anpassung an die konkrete Maschine vor. Zwei Bemerkungen dazu:

1. Dieser Eindruck ist richtig.
2. Die Schlussfolgerung, dass jedes für Maschinenabhängigkeiten nur diese Sprachelemente benutzende Programm portabel sei, ist falsch.

Es kann sein, dass ein auf einer Maschine funktionierendes Programm vom MODULA-2-Compiler einer anderen Maschine als "nichtanpassbar" zurückgewiesen wird. Als Beispiel sei nur die Frage genannt, ob ein aktueller Parameter vom Typ 'ARRAY [0..5] OF CHAR' auf der Position eines 'ARRAY-OF-WORD'-Parameters akzeptiert wird. Belegt er 6 oder 3 oder nur anderthalb Speicherwörter?

Es kann aber auch sein, dass der Compiler die Anpassung an eine neue Maschine problemlos bewältigt, die Abarbeitung jedoch sehr "falsch" läuft. "Falsch" ist hier i. allg. falsch: Man hat beim Programmieren versehentlich auf Verhältnisse vertraut, die in der neuen Umgebung (Computer, Betriebssystem, MODULA-2-System) nicht mehr gelten.

Ich gebe Ihnen ein einfaches Beispiel an. In der Prozedur 'Vertau-

schen' habe ich stillschweigend vorausgesetzt, dass die Längen der beiden Parameter übereinstimmen. Gegeben seien zwei Variablen.

```
VAR
   ii: ARRAY [0..3] OF INTEGER;
   rr: RECORD r1,r2: OF REAL END;
```

Bei meinem MODULA-2-System arbeitet 'Vertauschen(ii,rr)' korrekt. Aber in [Logi85] passiert grosser Unfug; hier nimmt nämlich die Variable 'rr' doppelt soviel Speicherplatz wie 'ii' ein.
Als vor- und weitsichtiger Programmierer könnte man in 'Vertauschen' ganz vorn eine Zeile

```
IF HIGH(a)#HIGH(b) THEN HALT END;
```

einfügen. Dann hat man zwar ein Programm, das in der einen Umgebung läuft und in der anderen abstürzt, aber man wird nachhaltig auf das Problem aufmerksam gemacht.
Die Prozeduren 'NEWPROCESS' und 'TRANSFER' dienen zum Einrichten einer Coroutine bzw. zum Umschalten zwischen zwei Coroutinen (vgl. die Bemerkungen im Abschn. 1.2).
Jeder Typbezeichner 'T' ist verwendbar als Typumwandlungsfunktion

```
PROCEDURE T(x: ...): T.
```

Bei Funktionsaufrufen 'T(a)' muss der Typ 'TA' des als aktueller Parameter notierten Ausdrucks 'a' der Bedingung

```
TSIZE(TA) = TSIZE(T)
```

genügen. Derartige Aufrufe lösen keine Laufzeitaktion aus. Sie veranlassen lediglich den Compiler, den Ausdruck 'a' so zu interpretieren, als wäre er vom Typ 'T'.
Ich erinnere daran, dass ähnliche Typumwandlungen implizit auch bei Wertzuweisungen (vgl. Abschn. 8.1) und mittels trickreicher Records (s. Programm 9.1) möglich sind. Bitte beherzigen Sie die in diesen Zusammenhängen ausgesprochenen Warnungen, und seien Sie auch mit der Verwendung expliziter Typumwandlungen vorsichtig!
Ich mache von der Möglichkeit, Typbezeichner als Typumwandlungsfunktionen zu verwenden, nur in zwei Standardsituationen Gebrauch, wo CARDINAL-Werte als INTEGER-Zahlen interpretiert werden sollen:

1. Resultat der Funktion 'ORD' beim Einlesen von Zahlen
2. Vergleich des Resultats von 'HIGH' mit INTEGER-Indizes.

Damit glaube ich, alles Wesentliche über MODULA-2 gesagt zu haben, und wiederhole: Die Sprache MODULA-2 gefällt mir.

Anhang: Syntax von MODULA-2

```
&    bezeichner = buchstabe { buchstabe | ziffer }
&    buchstabe  = "a" | "b" |     | "z" | "A" | "B" |     | "Z"
&    ziffer     = "0" | "1" |     | "9"
&    zahl = ganze_zahl | reelle_zahl
&    ganze_zahl  = ziffer {ziffer}
&                | oktalziffer {oktalziffer} "B"
&                | ziffer{hexziffer} "H"
&    oktalziffer = "0" | "1" |     | "7"
&    hexziffer   = ziffer | "A" | "B" |     | "F".
&    reelle_zahl = ziffer {ziffer} "." {ziffer} [skalierung]
&    skalierung  = "E" ["+"|"-"] ziffer{ziffer}
&    zeichenkette = "'" {zeichen_ausser_apostroph}
&                 |     {zeichen_ausser_doppelapostroph}
&                 | oktalziffer {oktalziffer} "C"
&    kommentar = "(*" { textstueck | kommentar } "*)"

$    programmodul = MODULE bezeichner ";"
$                   {import ";"} block bezeichner "."
$    import = IMPORT liste_von_modulnamen
$           | FROM modulname IMPORT bezeichnerliste
$    liste_von_modulnamen = bezeichnerliste .
$    modulname = bezeichner
$    block = {vereinbarung} [BEGIN anweisungsfolge] END
$    anweisungsfolge = anweisung {";" anweisung}
$    variablenvereinbarung = bezeichnerliste ":" typ
$    bezeichnerliste = bezeichner {"," bezeichner}
$    konstantenvereinbarung = bezeichner "=" konstantenausdruck .
$    prozedurvereinbarung = prozedurkopf ";" block bezeichner
$    prozedurkopf = PROCEDURE bezeichner signatur
$    signatur = [ "(" [f_par_liste] ")" [":" typname] ]
$    f_par_liste = f_par_abschnitt {";" f_par_abschnitt}
$    f_par_abschnitt = [VAR] bezeichnerliste ":" formaler_typ .
$    formaler_typ = [ARRAY OF] typname
$    typname = q_name
$    abstrakte_signatur = [ "(" [f_typ_liste] ")" [":" typname] ]
```

```
$    f_typ_liste = f_typ_angabe {"," f_typ_angabe}
$    f_typ_angabe = [VAR] formaler_typ .
$    rel_op = "<" | "<=" | ">" | ">=" | "=" | "<>" | "#" | IN
$    add_op = "+" | "-" | OR .
$    mul_op = "*" | "/" | DIV | MOD | AND | "&"
$    neg_op = "~" | NOT
$    ausdruck = einfacher_ausdruck [rel_op einfacher_ausdruck]
$    einfacher_ausdruck = ["+"|"-"] term {add_op term}
$    term = faktor {mul_op faktor}
$    faktor = elementaroperand | "(" ausdruck ")" | neg_op faktor
$    elementaroperand = zahl | zeichenkette | menge
$                     | bezugnahme | funktionsaufruf
$    konstantenausdruck =
$      einfacher_konst_ausdruck [rel_op einfacher_konst_ausdruck]
$    einfacher_konst_ausdruck =
$      ["+"|"-"] konst_term {add_op konst_term}
$    konst_term = konst_faktor {mul_op konst_faktor}
$    konst_faktor = el_konst_operand
$                 | "(" konstantenausdruck ")"
$                 | neg_op konst_faktor
$    el_konst_operand = zahl | zeichenkette | konst_menge
$                     | q_name
$    funktionsaufruf = bezugnahme aktuelle_parameter
$    aktuelle_parameter = "(" [akt_par_liste] ")"
$    akt_par_liste = akt_parameter {"," akt_parameter}
$    akt_parameter = ausdruck | bezugnahme
$    prozeduraufruf = bezugnahme [aktuelle_parameter]
$    anweisung = [ einfache_anweisung | with_anweisung
$                | anweisungsauswahl | zyklusanweisung ]
$    einfache_anweisung = wertzuweisung | prozeduraufruf
$                       | exit_anweisung | return_anweisung
$    wertzuweisung = bezugnahme ":=" ausdruck .
$    exit_anweisung = EXIT
$    return_anweisung = RETURN ausdruck | RETURN
$    anweisungsauswahl = if_anweisung | case_anweisung
$    if_anweisung = IF ausdruck THEN anweisungsfolge
$                   { ELSIF ausdruck THEN anweisungsfolge }
$                   [ ELSE anweisungsfolge ] END
```

```
$    case_anweisung = CASE ausdruck OF fall {"|" fall}
$                     [ ELSE anweisungsfolge ] END
$    fall = [ liste_von_fallkennzeichen ":" anweisungsfolge ]
$    liste_von_fallkennzeichen =
$                     fallkennzeichen {"," fallkennzeichen}
$    fallkennzeichen =
$              konstantenausdruck [".." konstantenausdruck]
$    zyklusanweisung = loop_anweisung | while_anweisung
$                    | repeat_anweisung | for_anweisung
$    loop_anweisung = LOOP anweisungsfolge END
$    while_anweisung = WHILE ausdruck DO anweisungsfolge END
$    repeat_anweisung = REPEAT anweisungsfolge UNTIL ausdruck
$    for_anweisung = FOR laufvariable ":=" anfangswert TO endwert
$                    [BY schrittweite] DO anweisungsfolge END
$    laufvariable = bezeichner
$    anfangswert = ausdruck
$    endwert = ausdruck .
$    schrittweite = konstantenausdruck
$    typ = q_name | diskreter_typ | record_typ | array_typ
$        | set_typ | prozedur_typ | pointer_typ .
$    typvereinbarung = bezeichner "=" typ .
$    vereinbarung = TYPE {typvereinbarung ";"}
$                 | VAR {variablenvereinbarung ";"}
$                 | CONST {konstantenvereinbarung ";" }
$                 | prozedurvereinbarung ";"
$                 | modulvereinbarung ";"
$    diskreter_typ = aufzaehlungstyp | teilbereichstyp .
$    aufzaehlungstyp = "(" bezeichnerliste ")"
$    teilbereichstyp = [q_name] teilbereich .
$    teilbereich = "[" untere_grenze ".." obere_grenze "]"
$    untere_grenze = konstantenausdruck .
$    obere_grenze = konstantenausdruck .
$    record_typ = RECORD liste_von_feldlisten END .
$    liste_von_feldlisten = feldliste {";" feldliste}
$    feldliste = [ bezeichnerliste ":" typ | variantenliste ]
$    bezugnahme = q_name {selektor}
$    selektor = record_selektor  | array_selektor
$             | pointer_zugriff
```

```
$    record_selektor = "." bezeichner
$    with_anweisung = WITH bezugnahme DO anweisungsfolge END .
$    variantenliste = CASE [bezeichner] ":" q_name OF
$                       variante {"|" variante}
$                     [ ELSE liste_von_feldlisten ] END .
$    variante =
$        [liste_von_fallkennzeichen ":" liste_von_feldlisten]
$    array_typ = ARRAY indextyp {"," indextyp} OF typ .
$    indextyp = q_name | diskreter_typ .
$    array_selektor = "[" index {"," index} "]"
$    index = ausdruck .
$    set_typ = SET OF basistyp .
$    basistyp = q_name | diskreter_typ
$    menge = [q_name] "{" [element {"," element}] "}"
$    element = ausdruck [".." ausdruck]
$    konst_menge =
$           [q_name] "{" konst_element {"," konst_element} "}" .
$    konst_element = konstantenausdruck [".." konstantenausdruck]
$    prozedur_typ = PROCEDURE abstrakte_signatur
$    uebersetzungseinheit = programmodul
$                         | definitionsmodul
$                         | implementationsmodul
$    definitionsmodul = DEFINITION MODULE bezeichner ";"
?                       EXPORT QUALIFIED bezeichnerliste ";"
$                       {import ";"} {definition}
$                       END bezeichner "."
$    implementationsmodul = IMPLEMENTATION programmodul
$    definition = CONST {konstantenvereinbarung ";"}
$               | VAR {variablenvereinbarung ";"}
$               | TYPE { (typvereinbarung | bezeichner) ";"}
$               | prozedurkopf ";"
$    pointertyp = POINTER TO typ.
$    pointer_zugriff = "^".
$    modulvereinbarung = MODULE bezeichner [prioritaet] ";"
$                         {import ";"} [export] block bezeichner .
$    prioritaet = "[" konstantenausdruck "]"
$    export = EXPORT [QUALIFIED] bezeichnerliste
$    q_name = {modulname "."} bezeichner
```

Literaturverzeichnis

[Ahre85] Ahrens, K.: Modellierung und Simulation in MODULA-Programmierumgebungen. Berlin: Dissertation Humboldt-Universität 1985

[Amma75] Ammann, U.: Die Entwicklung eines PASCAL-Compilers nach der Methode des Strukturierten Programmierens. Zürich: Dissertation ETH 1975

[Amma78] Ammann, U.: Error recovery in recursive descent parsers. ETH Zürich, Institut für Informatik, Bericht 25, Mai 1978

[Blac85] Blach, R.: Implementationen von Modula-2 für 8- und 16-Bit-Prozessoren. edv-aspekte 4(1985) H. 3, S. 9-11

[Both78] Bothe, K.: Spezifikation und Verifikation abstrakter Datentypen. Berlin: Diss. Humboldt-Universität 1978

[EnRe88] Engeln-Müllges, G.; Reutter, F.: Formelsammlung zur Numerischen Mathematik mit MODULA-2-Programmen. Mannheim, Wien, Zürich: BI-Wissenschaftsverlag 1988

[Gand85] Gander, W.: Computermathematik. Basel, Boston, Stuttgart: Birkhäuser Verlag 1985

[Gand86] Gander, W.: Computermathematik - Lösungen der Aufgaben mit Turbo-PASCAL-Programmen. Basel, Boston, Stuttgart: Birkhäuser Verlag 1986

[Geis81] Geissmann, L.: A User Guide to the Modula-2 System M2RT11. ETH Zürich, Institut für Informatik, September 1981

[Hoar62] Hoare, C. A. R.: Quicksort. The Computer Journal 5(1962) H. 1, S. 10-15

[HoSa81] Horowitz, E.; Sahni, S.: Algorithmen - Entwurf und Analyse. Berlin, Heidelberg, New York: Springer-Verlag 1981

[KiSc88] Kielbasinski, A.; Schwetlick, H.: Numerische lineare Algebra. Berlin: VEB Deutscher Verlag der Wissenschaften 1988

[Knau87] Knauf, R.: Untersuchungen über Aufbau und Implementierung von Expertensystemen und Realisierung eines Diagnosesystems in der Programmiersprache PROLOG. Technische Hochschule Ilmenau, Sektion Techn. und Biomed. Kybernetik, Technischer Bericht, Juni 1987

[LiTr87] Lindner, U.; Trautloft, R.: Grundlagen der problemorientierten Programmentwicklung. Berlin: VEB Verlag Technik 1987

[Logi85] MODULA-2/86 User's Manual. LOGITECH, Inc., Redwood City, LU-GU101-3, December 1985

[Logi86] MODULA-2/86 Make Utility. LOGITECH, Inc., Redwood City, June 1986

[Polz79] Polze, C.: Implementation eines CDL-PASCAL-Übersetzers. Wiss. Zeitschrift der Humboldt-Universität zu Berlin, Math.-Nat. Reihe XXVIII(1979) H.4, S. 567-573

[Pomb84] Pomberger, G.: Softwaretechnik und MODULA-2. München, Wien: Oldenbourg Verlag 1984

[Robo87a] Programmiersystem Modula-2, AC A7100/A7150/EC1834, Sprachbeschreibung. VEB Robotron-Projekt Dresden, C1016-0900-1 M3030, März 1987

[Robo87b] Programmiersystem Modula-2, AC A7150/EC1834, Anleitung für den Bediener (DCP). VEB Robotron-Projekt Dresden, C3015-0900-1 M3030, November 1987

[Wilk79] Wilkinson, J. H.: Rundungsfehler. Berlin, Heidelberg, New York: Springer-Verlag 1969

[Wirt77] Wirth, N.: Compilerbau. Stuttgart: B. G. Teubner Verlag 1977

[Wirt80] Wirth, N.: MODULA-2. 2. Auflage. ETH Zürich, Institut für Informatik, Bericht 36, Dezember 1980

[Wirt82] Wirth, N.: Programming in Modula-2 (with defining report). Berlin, New York: Springer-Verlag 1982

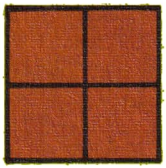
[Wirt84] Wirth, N.: Revisions and amendments to MODULA-2. Zürich: Institut für Informatik der ETH, 24. Mai 1984

[Wirt86] Wirth, N.: Algorithms and data structures. 2nd edition. London: Prentice-Hall International 1986

[Wirt87] Wirth, N.: From Modula to Oberon and The Programming Language Oberon. ETH Zürich, Institut für Informatik, Bericht 82, September 1987

[Ziel78] Zielinski, R.: Erzeugung von Zufallszahlen. Leipzig: VEB Fachbuchverlag 1978

Sachwörterverzeichnis



Die Programmiersprache MODULA-2 ist eine Weiterentwicklung von PASCAL und gewinnt national und international an Bedeutung. Ihr Wert liegt im Bausteinkonzept.
In dem Buch wird die Sprache MODULA-2 vollständig vorgestellt.
Die MODULA-2-Sprachelemente werden durch eine Vielzahl von getesteten Beispielen illustriert.

ISBN 3-341-00857-8