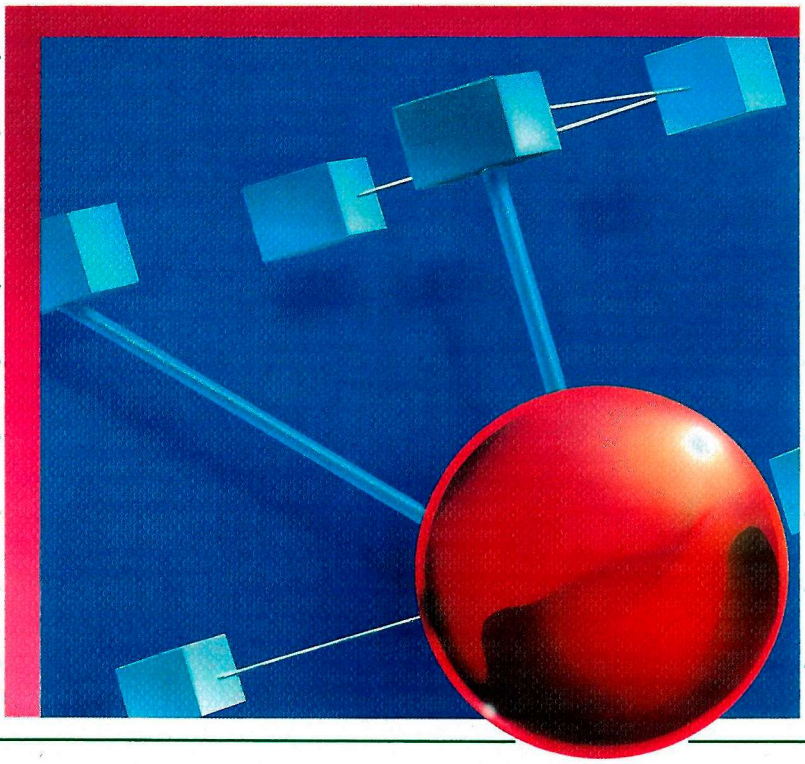
Ulrich Kern

Wie man in Modula-2 programmiert



CHIP WISSEN

Ein Buch von CHIP, dem Mikrocomputer-Magazin

Modula-2 ist die Programmiersprache der Zukunft. Aufgrund ihres geringen Sprachumfanges ist sie hervorragend zum Erlernen der Programmierung geeignet. Darüber hinaus erfüllt das bereitgestellte Baukastenprinzip alle Anforderungen der modernen Software-Entwicklung.
In Modula-2 geschriebene Programme sind in hohem Maß portierbar, d. h. mit wenig Aufwand auf verschiedenen Rechnern einsetzbar. Modula-2-Compiler gibt es für nahezu jeden Computer, vom PC bis zum Großrechner.
Das Buch enthält einen vollständigen Einführungskurs in das Programmieren mit Modula-2. Die einzelnen Sprachkonzepte werden anhand zahlreicher Beispiele dargestellt. Übungen und deren exemplarische Lösungen vertiefen den behandelten Stoff. Der Leser lernt den Umgang mit Algorithmen und Datenstrukturen, von einfachen Anweisungen bis hin zu Rekursion und Zeigertypen.

VOGEL-BUCHVERLAG
WÜRZBURG

ISBN 3-8023-0207-9

4 003634 002071

DM 40,– 1. Auflage

Ulrich Kern

Wie man in Modula-2 programmiert

VOGEL Buchverlag Würzburg

Ulrich Kern
Jahrgang 1952; 1973–1984 Studium der Logik, Wissenschaftstheorie, Philosophie, Germanistik und Theaterwissenschaft an der Ludwig-Maximilians-Universität in München. 1985 freiberuflicher Dozent für Programmierkurse in Pascal und Modula-2. Seit 1987 stellvertretender Chefredakteur von CHIP SPECIAL.

Für Christine, Thomas und Jessica

CIP-Titelaufnahme der Deutschen Bibliothek

Kern, Ulrich:
Wie man in Modula-2 programmiert / Ulrich Kern. – Würzburg: Vogel, 1989
ISBN 3-8023-0207-9

ISBN 3-8023-0207-9
1. Auflage 1989

Printed in Germany

Umschlaggrafik: Michael M. Kappenstein, Frankfurt/Main
Herstellung: Alois Erdl KG, Trostberg

Vorwort

Das vorliegende Buch gibt eine vollständige Einführung in die Programmierung mit Modula-2. Es ist in erster Linie zum Selbststudium gedacht, kann aber auch als Begleitliteratur zu Kursen verwendet werden. Der Leser muß keine speziellen Vorkenntnisse besitzen. Auf eine Beschreibung dessen, was ein Computer ist und wo seine Einsatzbereiche liegen, wurde allerdings verzichtet, da solche Informationen inzwischen zum Allgemeinwissen zählen.

Der behandelte Stoff geht beträchtlich über elementare Sprach- und Programmierkonzepte hinaus. So wird rekursiven Algorithmen und dynamischen Strukturen dieselbe Aufmerksamkeit gewidmet wie einfachen Anweisungen oder selbstdefinierten Typen. Verzichtet wurde in der Hauptsache nur auf Aspekte der Systemprogrammierung, deren Techniken oft im Widerspruch zu der didaktischen Forderung nach einem sauberen Stil stehen.

Die zahlreichen Beispiele und Aufgaben dienen einem doppelten Zweck: Einerseits sollen sie die beim Selbststudium fehlende Lehrer-Schüler-Kommunikation ersetzen, soweit das im Rahmen eines Buches überhaupt möglich ist. Andererseits sollen sie eine Brücke schlagen von der systematischen Gliederung des Stoffes zu den Fortschritten in der Programmierkunst.

Einen besonderen Schwerpunkt bilden die Programme zur Textanalyse und -bearbeitung. Der Grund hierfür ist, daß an diesen Beispielen so grundlegende Programmiertechniken wie die Ein- und Ausgabe von Daten oder die Konstruktion endlicher Automaten klargemacht werden können. Daß bei der Programmierarbeit das Datenmaterial in Form von Quelltexten ganz automatisch anfällt, ist ein günstiger Nebeneffekt.

Das Buch basiert in der Hauptsache auf einer Reihe von Programmierkursen, die ich im Verlauf der letzten Jahre in München gehalten habe. Den Teilnehmern möchte ich an dieser Stelle ganz ausdrücklich danken. Sie haben mit beharrlichen Fragen und Anregungen eine Menge zur Aufbereitung des Stoffes beigetragen. Viele

Selbstverständlichkeiten erweisen sich bei näherem Hinsehen als komplizierte Gebilde. Und vieles, von dem man glaubt, es sei nur Experten zugänglich, beherrschen Anfänger bereits, ohne auch nur ein Wort darüber zu verlieren.

Die Programmbeispiele stellen keine besonderen Ansprüche an den verwendeten Modula-2-Compiler. Einzige Voraussetzung ist, daß die Module «InOut», «RealInOut», «Terminal» und «Storage» dem Standard nach Niklaus Wirth entsprechend implementiert sind.

Eine Diskette (MS-DOS, 360 KB, DS/DD; andere Formate auf Anfrage) mit den Programmen dieses Buches ist zum Preis von 35 DM zuzügl. 3 DM Versandkosten erhältlich bei: Vogel-Buchverlag, Günter Rolle, Schillerstraße 23a, 8000 München 2, Telefon: (089) 5 14 93 33.

München Ulrich Kern

Inhaltsverzeichnis

1 Programmieren – was ist das?

Ein Computer ohne Programm ist ein nutzloser Gegenstand. Erst das Programm macht ihn zur Maschine. Durch die Fähigkeit, verschiedene Programme ausführen zu können, wird er zur Universalmaschine im Bereich der Datenverarbeitung.

Das Erstellen eines Computerprogramms ist daher mit der Konstruktion einer speziellen Maschine vergleichbar, bei der anstelle der Bearbeitung irgendwelcher Werkstoffe die Manipulation von Informationen tritt. Es ist also durchaus angebracht, beim Programmierer von einem Software-Ingenieur zu sprechen. Programmieren heißt in diesem Sinne: eine Datenverarbeitungsmaschine bauen.

Die Vorgehensweise dabei läßt sich wie folgt skizzieren: Zunächst wird festgelegt, was die Maschine leisten soll. Dann werden funktionale Einheiten entworfen, deren Zusammenspiel die Gesamtleistung bewirkt. In manchen Fällen kann auf bereits Vorhandenes zurückgegriffen werden. Es ist sicher sinnvoll, bewährte Komponenten zu verwenden und die neu zu schaffenden entsprechend anzupassen. Je nach Komplexität der so festgelegten Funktionseinheiten werden diese isoliert betrachtet und einer ähnlichen Aufteilung unterworfen wie die gesamte Maschine. Dieses Verfahren wird so lange wiederholt, bis die einzelnen Komponenten klar, einfach und sicher realisiert werden können. Erst jetzt wird das Stadium der Planung und Konstruktion verlassen, um die konkrete Ausführung in Angriff zu nehmen. Beim anschließenden Zusammenbau der Komponenten muß das reibungslose Zusammenspiel laufend kontrolliert werden. Häufig machen sich dabei Konstruktionsfehler bemerkbar, deren Behebung mitunter zu einer Revidierung kompletter Konstruktionswege führt.

Die größte Fehlerquelle steckt jedoch in der konkreten Ausführung. So wie unpassende Gewinde, falsch bemessene Abstände, schlechte Materialauswahl etc. einen nach außen hin vielleicht funktionstüchtigen Apparat zum Gefahrenherd werden lassen, können in der Datenverarbeitung ein paar unbedacht eingesetzte

Programmschritte oder schlecht gewählte Größen fatale Auswirkungen haben. Aus diesem Grund ist es von höchster Wichtigkeit, bei der konkreten Ausführung über leistungsfähige Werkzeuge zu verfügen, mit deren Hilfe die möglichen Fehlerquellen auf ein Minimum reduziert werden können.

1.1 Die Programmiersprache Modula-2

Das wichtigste Werkzeug eines Programmierers ist die Programmiersprache. Mit ihr werden die gewünschten Arbeitsabläufe auf den Computer übertragen. Es dürfte klar sein, daß an die Qualität einer Programmiersprache höchste Anforderungen gestellt werden. Eine Programmiersprache ist um so besser, je mehr sie die Aufteilung in funktionale Einheiten (Modularisierung), eine klare und übersichtliche Darstellung der Abläufe (Strukturierung) und eine unmittelbare und logische Bearbeitung realer Größen unabhängig von deren computerinterner Darstellung (Abstraktion) ermöglicht. Das sind die Anforderungen, die von der Seite der Anwenderprogrammierung an eine Programmiersprache gestellt werden. Soll die Sprache auch für die Systemprogrammierung geeignet sein, so müssen zusätzlich sogenannte «niedere Sprachelemente» vorhanden sein, über die auf die einzelnen physikalischen Komponenten des Computers zugegriffen werden kann.

Es wäre jedoch falsch anzunehmen, daß eine Programmiersprache erst bei der konkreten Ausführung einer Programmkonstruktion ins Spiel kommt. Obwohl man in Informatikerkreisen lange Zeit dieser Ansicht war, hat sich gezeigt, daß die jeweils eingesetzte Programmiersprache bereits direkt in die Planungsarbeit einwirkt. Die günstigste Wechselwirkung besteht dann, wenn die verwendete Programmiersprache Mittel zur Verfügung stellt, mit der sich Konstruktionspläne direkt ausdrücken lassen.

Modula-2 ist eine moderne Programmiersprache, die alle vorgenannten Forderungen in geradezu vorbildlicher Weise erfüllt. Sie wurde an der ETH Zürich vom Schweizer Informatiker Niklaus Wirth entwickelt und 1983 mit dem Buch «Programming in Modula-2» der Öffentlichkeit vorgestellt. Bei den großen Universalsprachen ist sie der mächtige Konkurrent von Ada, einer vom nordamerikanischen Verteidigungsministerium initiierten Programmiersprache. Die besonderen Vorzüge von Modula-2 beruhen auf der wohlüberlegten Bemessung des Sprachumfangs. So sind alle

Elemente vorhanden, die zum Realisieren auch größter Programmprojekte benötigt werden. Andererseits ist die Anzahl der Sprachkonstrukte so gering, daß Modula-2 mit Recht zu den leicht erlernbaren Programmiersprachen zählt. Auch dem Anfänger, der noch keine Programmiersprache beherrscht, kann Modula-2 uneingeschränkt empfohlen werden. Er erwirbt damit ein Werkzeug, das ihn mehr als alle anderen darin unterstützt, gute Programme zu schreiben.

1.2 Gute Programme

Was ist ein gutes Programm? Welche Kriterien müssen erfüllt sein, daß man von einer gelungenen Maschine sprechen kann? Die Aufzählung kann durchaus als verbindlich betrachtet werden:

- ☐ Erbringen der gewünschten Leistungen,
- ☐ Fehlerfreiheit,
- ☐ Bedienungsfreundlichkeit,
- ☐ Effizienz,
- ☐ Ästhetik.

Das sind die Kriterien, nach denen ein Anwender ein Computerprogramm beurteilt. Keiner der Punkte ist leicht zu erfüllen. Man kann getrost davon ausgehen, daß in jedem etwas komplexeren Programm mehrere Fehler vorhanden sind. Das bedeutet, daß es unter ganz bestimmten Umständen falsche Resultate liefert. Die große Gefahr liegt besonders darin, daß unkorrekte Ergebnisse oftmals nicht erkannt werden können (vor allem, wenn das Programm normalerweise zufriedenstellend arbeitet). Es steht fest, daß die Fehlersuche der aufwendigste Teil der Programmierarbeit ist.

Überraschend in diesem Zusammenhang ist vielleicht die Forderung nach Ästhetik. Sie erklärt sich aus der Tatsache, daß die Steuerung moderner Anwenderprogramme grundsätzlich dialogorientiert am Bildschirmterminal stattfindet. Es würde einem kulturellen Rückschritt gleichkommen, würde man der Gestaltung dieser Schnittstelle zwischen Mensch und Computer ausschließlich funktionalen Wert beimessen. Das Design einer Maschine ist ein wesentlicher Faktor, der freilich erst dann zum Tragen kommt, wenn die Maschine ordnungsgemäß funktioniert.

Auf der Programmiererseite besteht noch eine weitere Forderung an ein gutes Programm: Sein Konstruktionsplan muß klar und

nachvollziehbar sein. Nur dann können Fehler lokalisiert, neue Leistungen eingebaut oder bestehende ergänzt und verändert werden. Komplizierte Gedankengänge müssen ausreichend dokumentiert sein, auf undurchsichtige Tricks wird grundsätzlich verzichtet. Auch hier spielt die verwendete Programmiersprache eine große Rolle. Wenn sich der Entwurf im Programmtext direkt widerspiegelt, ist in bezug auf Nachvollziehbarkeit schon sehr viel gewonnen.

1.3 Editor, Compiler und Linker

Eine Programmiersprache ist, im Vergleich zu den natürlichen Sprachen, ein sehr primitives Kommunikationsmittel. Nur einige wenige Wörter und Symbole werden zur Darstellung von Arbeitsabläufen bereitgestellt. Zudem funktioniert die Verständigung mit einer Programmiersprache nur in einer Richtung. Der Programmierer formuliert Sätze in Form von eindeutigen Befehlen, die der Computer dann ausführt. Nun kann aber auch ein Computer mit den Sätzen einer Programmiersprache nicht unmittelbar etwas anfangen. Sie müssen erst in eine ihm direkt verständliche Form übersetzt werden. Diese Übersetzung übernimmt normalerweise ein spezielles Programm mit dem Namen «Compiler» (Kompilierer). Um einer Sprachverwirrung zuvorzukommen, unterscheidet man den Quelltext eines Programms, den der Programmierer formuliert, vom sogenannten Maschinencode, den der Computer ausführen bzw. abarbeiten kann. Während der Programmierer den Begriff «Programm» mehr mit dem Quelltext verbindet, tritt dem Anwender «das Programm» in Form des gerade bearbeiteten Maschinencodes gegenüber.

Auch zum Formulieren des Quelltextes gibt es Hilfsmittel, sogenannte Editoren oder Textverarbeitungssysteme. Damit kann der Programmtext direkt am Computerterminal erstellt werden. Zu der komfortablen Bearbeitung – Löschen, Einfügen, Verschieben oder Kopieren von Zeichen und Textblöcken – kommt noch der Vorteil, daß der Text so im Computer abgelegt wird, daß er ohne weitere Manipulation vom Compiler bearbeitet werden kann. In Modula-2 kann bereits der Compiler viele mögliche Fehler erkennen und durch geeignete Meldungen darauf hinweisen. Wenn der Compiler den Quelltext ohne Fehlermeldung übersetzt hat, muß der entstandene Maschinencode meist noch mit dem von anderen

benötigten Programmteilen zusammengebunden werden. Das Programm, das diese Arbeit übernimmt, heißt Linker oder Binder.

Als erstes sollte man sich mit diesen drei Programmen vertraut machen. Da die Arbeitsweise von System zu System unterschiedlich ist, bleibt nichts anderes als ein Studium der entsprechenden Handbücher oder, was unbedingt vorzuziehen ist, die Befragung eines Bekannten, der mit dem System bereits vertraut ist. Über folgende Fähigkeiten sollte man unbedingt verfügen, damit die Beispiele und Aufgaben des Kurses auch praktisch nachvollzogen werden können (verwenden Sie hierfür den Text des Moduls «Hallo»):

- ☐ Eingeben eines Programmtextes mit Hilfe eines Editors,
- ☐ Modifizieren eines bereits abgespeicherten Textes,
- ☐ Starten des Compilers zum Übersetzen des Programmtextes,
- ☐ Starten des Linkers, um ein ausführbares Programm zu erhalten,
- ☐ Starten des erzeugten Programms.

2 Das erste Programm

Schauen wir uns erst einmal das allereinfachste Modula-2-Programm an:

```
MODULE LeeresProgramm;
END LeeresProgramm.
```

Man erkennt zunächst die beiden fettgedruckten Wörter «MODULE» und «END». Das sind sog. Schlüsselwörter oder auch reservierte Wörter. Sie haben eine genau festgelegte Bedeutung und können niemals umdefiniert werden. Der Fettdruck dieser Wörter sorgt für einen besseren Überblick im Programm. Ein Programm, das einen beliebigen Modula-2-Text in der dargestellten Form ausdruckt, werden wir später selbst erarbeiten.

An dem obigen Beispiel kann bereits die Grundstruktur eines Modula-2-Programms erklärt werden: Am Anfang steht immer das Schlüsselwort «MODULE». Darauf folgt offensichtlich der Name des Programms, in unserem Beispiel «LeeresProgramm». Im Anschluß an diesen Namen folgt ein Strichpunkt. Dann kommt wieder ein Schlüsselwort, nämlich «END», und im Anschluß daran nochmals der Programmname mit abschließendem Punkt.

Zum Vereinfachen der Syntaxdarstellung des Programmaufbaus gibt es einige sehr einfache Abkürzungen und Regeln. Nach diesen wird die Grundstruktur so ausgedrückt:

Programm-Modul::="MODULE" Modulname ";" Block Modul-Name ".".
Block::="END".

Nach dieser Schreibweise werden sprachliche Ausdrücke, die Bestandteil der Sprache sind (wie etwa das reservierte Wort «MODULE» oder der Strichpunkt) in Anführungszeichen geschrieben. Das heißt, daß sie an der angegebenen Stelle auch genauso

stehen müssen. Der Ausdruck «::=» bedeutet soviel wie «definitionsgemäß gleich» und ist am einfachsten zu lesen als «... ist wie folgt aufgebaut». Da sich eine Definition auch über mehrere Zeilen erstrecken kann, wird ihr Ende mit einem Punkt gekennzeichnet. Somit ist die erste Zeile der Programm-Modul-Erklärung so zu lesen:

Ein Programm-Modul ist wie folgt aufgebaut: Auf das Schlüsselwort «MODULE» folgt der Modulname und dann ein Strichpunkt. Anschließend kommt ein Block, dann nochmals der Modulname und abschließend ein Punkt.

Kann man sich unter «Modulname» noch etwas vorstellen, so bedarf der Ausdruck «Block» einer weiteren Erklärung. Sie besagt, daß ein Block – im einfachsten Fall – aus dem reservierten Wort «END» besteht.

Allerdings muß zur exakten Darlegung des Aufbaus einer Programmiersprache jeder Ausdruck, der nicht in Anführungszeichen steht, weiter aufgelöst werden.

2.1 Bezeichner

Beim Modulnamen kann der eigenen Kreativität freier Lauf gelassen werden, solange folgende Regel beherzigt wird: Er muß stets mit einem Buchstaben beginnen, an den sich eine beliebige Anzahl von Buchstaben und Ziffern anschließt.

In der EBNF-(erweiterter Backus-Naur-Formalismus)Notation, wie unsere Syntax-Darstellungsmethode heißt, schaut das Ganze dann so aus:

```
Modulname::=Bezeichner.
Bezeichner::=Buchstabe{Buchstabe | Ziffer}.
Buchstabe::="A" | "B" ... | "Z" | "a" | "b" ... | "z".
Ziffer:="0" | "1" | "2" ... | "9".
```

Hier sind neue Zeichen aufgetreten: | bedeutet soviel wie «oder» bzw. «wahlweise». a | b heißt also, daß an dieser Stelle entweder a oder b stehen darf.

{ a } bedeutet, daß an dieser Stelle a beliebig oft stehen kann, also keinmal, einmal, zweimal usw.

Aus der obigen Syntaxbeschreibung für den Modulnamen, die übrigens für alle Namen gilt, die man bei der Programmerstellung

vergeben kann, folgt, daß ein gültiger Name im einfachsten Fall auch nur aus einem Buchstaben bestehen darf. Aber nicht alles, was erlaubt ist, ist auch sinnvoll. Folgende Namen sind bsw. erlaubt:

A
XYZ
A1B2C3
GrrPfrrzzGrhmm
WortStatistik

Zurückgewiesen werden hingegen:

1XYZ	(beginnt mit einer Ziffer)
Ab und Zu	(enthält Leerzeichen)
Ab_und_Zu	(auch der Unterstrich ist nicht zugelassen!)
END	(ist ein reserviertes Wort)

Aus der Beschreibung geht aber nicht hervor, wie lang die Namen sein dürfen, wie viele Zeichen also maximal zugelassen sind. Da jeder Text – und um einen solchen handelt es sich ja auch bei einem Programmtext – in Zeilen gegliedert ist und ein Name nicht über mehrere Zeilen gehen darf (denn dazu müßte er die Zeichen für «Neue Zeile» beinhalten, was nach obiger Vorschrift ausgeschlossen ist), setzt hier der verwendete Texteditor die Grenzen. Andererseits gibt es auch einige Modula-2-Compiler, bei denen die Anzahl der Zeichen begrenzt ist, nach denen die Namen unterschieden werden. Ist diese Anzahl kleiner als 20, so sollte dieser Compiler nicht verwendet werden, denn hier würde ein unerwünschter Zwang zu Abkürzungen eintreten.

Der großen Freiheit, die der Programmierer bei der Namensgebung genießt, steht die Forderung entgegen, daß ein Programmtext leichtverständlich sein soll. Man sollte sich erinnern, daß die höheren Programmiersprachen allein aus dem Grund geschaffen wurden, um die Verständigung zwischen Mensch und Computer zu vereinfachen. Auf der Rechnerseite werden vom Compiler alle Wendungen prompt zurückgewiesen, die dieser nicht versteht. Zeigen Sie dieselbe Konsequenz, indem Sie ausschließlich und immer «sprechende» Namen verwenden, damit jeder Mensch (sofern er der verwendeten Programmiersprache mächtig ist) Ihre Programme versteht.

Also nicht	MODULE A
oder	MODULE Bsp12
sondern	MODULE ZahlenSortierung
oder	MODULE QuickSortDemo

Noch ein Hinweis zu den Bezeichnern. Bei vielen Programmiersprachen wird die Groß-/Kleinschreibung ignoriert, nicht jedoch bei Modula-2. Hier sind «Zahlensortierung» und «ZahlenSortierung» verschiedene Bezeichner.

2.2 Kommentare

Als weiteres Hilfsmittel zum Programmverständnis durch den Menschen steht die Möglichkeit zur Verfügung, an jeder beliebigen Stelle eines Programms Kommentare einzufügen. In Modula-2 sieht ein Kommentar folgendermaßen aus:

(* beliebiger Text *)

Ein Kommentar wird also durch (* und *) gekennzeichnet. Er kann sich über beliebig viele Zeilen erstrecken. Zudem können Kommentare ineinander verschachtelt werden. Von dieser Möglichkeit wird man dann Gebrauch machen, wenn man ganze Programmteile ausblenden will. Unser kleines Programm könnte man bsw. so kommentieren:

```
MODULE LeeresProgramm; (* Das ist unser erstes Modula-2-
   Programm. Es ist mit Sicherheit absolut fehlerfrei, weil es
   nichts tut und somit keinen Schaden anrichtet. *)
END LeeresProgramm. (* Hier ist das Programm schon zu Ende *)
```

Viele Programmierapostel geben den Ratschlag: Fügen Sie so viele Kommentare wie möglich in Ihre Programme ein! Ich bin hier grundsätzlich anderer Meinung. Mein Rat: Kommentieren Sie so wenig wie nötig! Viel wichtiger sind klug gewählte Namen und eine klare optische Gliederung. Sie werden in den vielen Beispielen sehen, wann der Einsatz von Kommentaren sinnvoll und notwendig ist.

Noch ein Wort zur optischen Gliederung. Hier hat man in weitem Rahmen freie Hand. Das obige Programm könnte man auch so schreiben:

```
                                    MODULE
LeeresProgramm
;                                                    END
LeeresProgramm.
```

Oder so:

```
MODULE LeeresProgramm; END LeeresProgramm.
```

Jede Darstellung ist zugelassen, solange die Syntaxregeln nicht verletzt werden. Dem Compiler ist die optische Gestaltung eines Programms egal, sie dient ausschließlich der Lesbarkeit durch den Menschen. Es macht keinen Sinn, feste Regeln anzugeben, wann etwa eine Einrückung erfolgen muß oder eine neue Zeile begonnen werden sollte. Vielmehr sollen die Beispiele dieses Buches exemplarisch die Möglichkeiten der textlichen Gestaltung aufzeigen.

2.3 Ein- und Ausgabe am Terminal

Jetzt wollen wir den Rechner zu einer ersten Tätigkeit veranlassen: Er soll uns begrüßen, indem er den Satz «Hallo, hier bin ich!» auf dem Bildschirm ausgibt. Modula-2 enthält jedoch überhaupt keine Ein- und Ausgabeanweisung. Hier scheint ein Widerspruch vorzuliegen. Einerseits wurde Modula-2 als eine der modernsten Programmiersprachen bezeichnet, andererseits arbeiten alle modernen Programme interaktiv mit dem Benutzer. Dieser scheinbare Widerspruch ist jedoch schnell aufgeklärt. Die Ein- und Ausgabeanweisungen sind immer implementationsabhängig, d. h., daß sie für jeden Rechner ganz speziell gelöst werden müssen. Da die Sprache selbst jedoch unabhängig von speziellen Rechnertypen sein sollte, wurden alle diesbezüglichen Anweisungen aus dem Sprachkern ausgelagert. Sie sind in sog. Bibliotheksmoduln verfügbar, die zu jedem Modula-2-Compiler mitgeliefert werden.

Das vorläufig für uns wichtigste Bibliotheksmodul hat den Namen «InOut» und stellt u. a. folgende Prozeduren zur Verfügung: «WriteLn» und «WriteString». Die Funktion von «WriteLn» ist einfach zu erklären. Auf dem Bildschirm wird eine neue Zeile (engl. line) geschrieben, d. h., der Cursor (Leuchtzeiger) rückt an den Anfang der nächsten Zeile. Ist keine freie Zeile mehr auf dem Bildschirm vorhanden, wird der gesamte Inhalt um eine Zeile nach oben geschoben (Scrolling) und somit eine freie Zeile erzeugt.

Die Prozedur «WriteString» macht genau das, was ihr Name sagt: Sie schreibt einen String (Zeichenkette) auf den Bildschirm.

Der auszugebende String muß der Prozedur als Parameter (Argument) mitgegeben werden. Eine genaue Behandlung der Zeichenketten folgt erst später. Wir wollen an dieser Stelle nur auf Zeichenketten-Konstante eingehen, da das für den gewünschten Zweck ausreicht. Konstante sind Größen mit einem festen Wert. Der Satz «Hallo, hier bin ich!» ist eine feste Größe, also eine String-Konstante.

String-Konstante sind in Modula-2 beliebige Zeichenfolgen, die entweder von doppelten (Gänsefüßchen) oder einfachen (Hochkomma) Anführungszeichen eingeschlossen sind. Das einschließende Zeichen darf in der Zeichenfolge selbst nicht vorkommen. Beispiel für String-Konstante:

"Hallo, hier bin ich!"	
'Hallo, hier bin ich!'	
"Jetzt reicht's mir"	Wenn das einfache Anführungszeichen benötigt wird, nimmt man das doppelte zur Klammerung.
'Er sagte: "Es reicht!"'	Und umgekehrt.

Beispiele für unzulässige String-Konstante:

"Hallo, hier bin ich!'	Die Einschlußzeichen müssen gleich sein.
'Jetzt reicht's mir'	Das Einschlußzeichen darf selbst nicht vorkommen.

Jetzt stellt sich nur noch die Frage, wie wir an die gewünschten Prozeduren «WriteLn» und «WriteString» aus dem Modul «InOut» herankommen. Modula-2 stellt für diesen Zweck «IMPORT-Listen» zur Verfügung. Darin wird genau angegeben, welche Prozeduren aus welchem Modul eingebunden (importiert) werden. Die Importlisten folgen unmittelbar auf den Modul-Kopf und haben folgende Gestalt:

```
IMPORT-Listen::={"FROM" Modulname "IMPORT"
                    Bezeichner {"," Bezeichner} ";" }.
```

Das bedeutet also, daß ein Programm-Modul keine, aber auch beliebig viele Bezeichner aus anderen Moduln importieren kann. Wir können somit unsere Programmdefinition vervollständigen:

Programm-Modul::="MODULE" Modulname ";"
IMPORT-Listen Block ModulName ".".

2.4 Begrüßung durch den Computer

Jetzt wissen wir also, wie wir die benötigten Anweisungen in unser Programm bekommen. Unklar ist jedoch, wie wir sie dort verwenden können. Bei «WriteLn» und «WriteString» handelt es sich um Prozeduren, also selbständige Teilprogramme, die eine komplexe Aufgabe lösen (auch wenn es nicht den Anschein hat, die Bildschirmausgabe ist bereits eine komplizierte Aufgabe). Der Einsatz von Prozeduren geschieht ganz einfach durch Angabe des Prozedurnamens und, falls benötigt, der entsprechenden Parameter. Parameter werden immer in runden Klammern angegeben. Der Prozeduraufruf ist die erste Anweisung, die wir somit kennenlernen:

Anweisung::= Prozeduraufruf.
Prozeduraufruf::=Prozedurname ["(" Parameterliste ")"].
Parameterliste::=String-Konstante.

Hier taucht ein neues Zeichen in unserer formalen Sprache auf: Mit eckigen Klammern gekennzeichnete Teile können höchstens einmal auftreten, müssen es aber nicht.

Benötigt man in einem Programm mehrere Anweisungen, so werden sie – durch Strichpunkt (Semikolon) voneinander getrennt – nacheinander hingeschrieben. Sie werden dann in der angegebenen Reihenfolge abgearbeitet. In diesem Fall spricht man von einer Anweisungssequenz:

Anweisungssequenz::=Anweisung {";" Anweisung}.

Die Anweisungen selbst stehen innerhalb eines Blocks. Ein Block, so haben wir das am obigen Beispiel kennengelernt, kann nur aus dem reservierten Wort «END» bestehen. In diesem Fall handelt es sich um einen leeren Block. Jeder nichtleere Block beginnt mit dem Schlüsselwort «BEGIN». Dann folgt eine Anweisungssequenz und schließlich das Schlüsselwort «END». Somit ergibt sich für einen Block folgende Syntax:

Block::=["BEGIN" Anweisungssequenz] "END".

Mit diesen Vorüberlegungen können wir jetzt das gewünschte Programm leicht realisieren:

```
MODULE Hallo;
  FROM InOut IMPORT WriteLn, WriteString;
  BEGIN
     WriteLn;
     WriteString('Hallo, hier bin ich!');
     WriteLn
  END Hallo.
```

Aufgaben:

1. Was ist hier falsch?

   a)
   ```
   MODULE LeeresProgramm END Leeresprogramm;
   ```

   b)
   ```
   MODULE WoStecktDer_Fehler,
   FROM INOUT IMPORT WriteLn; WriteString;
      WriteLn; WriteString('Modula-2 mit Fehlern!")
      WriteLn
   END Hallo.
   ```

2. Aus welchen Symbolen besteht die EBNF-Notation, welche Bedeutung haben die Symbole?
3. Definieren Sie den Begriff «String-Konstante» mit Hilfe der EBNF-Notation.

3 Typen und Variable

Das Grundprinzip der Datenverarbeitung ist sehr einfach:

Eingabedaten –> Programm –> Ausgabedaten

Für die Ein- und Ausgabe von Daten stehen vielerlei Geräte zur Verfügung: Tastatur, Bildschirm, Disketten, Drucker, Scanner, Joysticks und vieles mehr. Allen diesen Geräten ist gemeinsam, daß sie immer nur einen Strom von Bits (binären Einheiten, 0–1-Zustände) übertragen. Um welche Art von Daten es sich dabei handelt – Zahlen, Buchstaben, Bilder etc. – wird erst durch das verarbeitende Programm definiert. Man kann sagen: Ein Programm interpretiert diese Daten als Buchstaben und jene als Karteikarten.

Um den verschiedenen Informationsarten des täglichen Lebens gerecht zu werden, stellt die Programmiersprache Modula-2 das Konzept der Datentypen zur Verfügung. Fest eingebaut sind dabei die am meisten benötigten Typen wie Zahlen und Buchstaben. Andere können beliebig zusammengestellt werden.

In diesem Kapitel werden die sog. Grundtypen behandelt, aus denen später komplizierte und dem jeweiligen Problem optimal angepaßte Strukturen zusammengesetzt werden können. Die Grundtypen sind im Sprachkern vorhanden, sie müssen nicht aus Bibliotheksmoduln importiert werden. Bezeichner, die zwar keine Schlüsselwörter, (wie «BEGIN») aber dennoch bereits im Sprachkern verfügbar sind, heißen «Standardbezeichner». Sie sind so aufzufassen, als wenn sie automatisch (aus einem unbekannten Modul) in jedes andere Modul importiert werden.

Die Grundtypen von Modula-2:

Name	Bedeutung	Beispiele
CARDINAL	Natürliche Zahlen inkl. 0	0 123 50000
INTEGER	Ganze Zahlen	−1000 0 125 32785

CHAR	Zeichen	A B C + - . !
REAL	Fließkommazahlen	3.14 -12.48E10
BOOLEAN	Wahrheitswerte	TRUE FALSE

Hinweis: Achten Sie auch bei den Standardbezeichnern darauf, daß diese (wie die Schlüsselwörter) immer in Großbuchstaben angegeben werden müssen.

3.1 Variablen-Deklaration

Ein weiterer, wichtiger Begriff ist die «Variable». Dabei handelt es sich um eine Größe, die verschiedene Werte annehmen kann (im Gegensatz zu einer Konstanten). Den Wert einer Variablen kann man abfragen, verändern oder zu Berechnungen verwenden. Man kann sagen: Variable sind das Gedächtnis eines Programms.

Betrachten wir einmal folgendes Problem: Wir sollen aus unserem Computer eine einfache Addiermaschine machen. Zu zwei eingegebenen Zahlen soll die Summe berechnet und ausgegeben werden. Ganz naiv, ohne gleich in einer Programmiersprache zu denken, könnte diese Aufgabe mit einer Anweisungsfolge gelöst werden:

1. Lies eine Zahl von der Tastatur.
2. Lies eine weitere Zahl.
3. Berechne die Summe aus den beiden Zahlen.
4. Gib die Summe auf dem Bildschirm aus.

Allerdings geht weder aus der Problemstellung noch aus dem Lösungsansatz hervor, um welche Zahlen es sich dabei handelt. Wir wollen annehmen, daß es sich um natürliche Zahlen handeln soll. Damit ist der Datentyp (CARDINAL) klar. Die erste Anweisung «Lies eine Zahl von der Tastatur» setzt bereits das Vorhandensein einer Variablen voraus, in der sich das Programm diese erste Zahl merken muß. Dasselbe gilt für die zweite Zahl und die Summe. Wir benötigen also drei Variable vom Typ CARDINAL. Dieser Bedarf wird dem Compiler im Deklarationsteil mitgeteilt. Um auf die einzelnen Variablen einfach zugreifen zu können, werden ihnen (wie dem Programm) Namen gegeben.

Beispiel:

```
VAR ErsteZahl, ZweiteZahl, Summe : CARDINAL;
```

Die Variablen-Deklaration beginnt also mit dem Schlüsselwort «VAR». Im Anschluß folgen, durch Komma getrennt, die Variablennamen und schließlich – nach einem Doppelpunkt – der Typ der Variablen.

Deklaration::=Variablen-Deklaration.
Variablen-Deklaration::="VAR" {Variablenliste ":" Typ ";"}.
Variablenliste::=VariablenName {"," VariablenName}.
VariablenName::=Bezeichner.

Der Deklarationsteil ist Bestandteil eines Blocks und kommt immer unmittelbar vor dessen Anweisungsteil.

Block::=[Deklaration] ["BEGIN" Anweisungssequenz] "END".

Vor diesem Hintergrund kann das obige Problem nun so angefaßt werden:

Benötigt werden die Variablen «ErsteZahl», «ZweiteZahl» und «Summe» vom Datentyp CARDINAL.

- ☐ Lies von der Tastatur die Variable «ErsteZahl».
- ☐ Lies ebenso die Variable «ZweiteZahl».
- ☐ Bestimme die Summe dieser beiden Variablen und weise das Ergebnis der Variablen «Summe» zu.
- ☐ Gib die Variable «Summe» auf dem Bildschirm aus.

Aber bevor wir diese Vorschrift nun endgültig in ein Programm übertragen können, müssen wir uns noch um die nötigen Ein- und Ausgabeprozeduren für CARDINAL-Zahlen kümmern. Sie sind ebenfalls im Modul «InOut» enthalten und heißen «ReadCard» und «WriteCard». Während bei «ReadCard» nur die einzulesende Variable als Parameter übergeben werden muß, bedarf «WriteCard» einer weiteren Angabe. Nach der auszugebenden Zahl wird, durch Komma getrennt, eine weitere (natürliche) Zahl als Formatangabe erwartet. Diese Zahl gibt die Breite des Feldes (in Zeichen) an, innerhalb dessen die auszugebende Zahl rechtsbündig ausgedruckt wird.

Mit diesen Informationen kann das Programm endlich in Angriff genommen werden:

```
MODULE Addition;
FROM InOut IMPORT ReadCard, WriteCard;
VAR ErsteZahl, ZweiteZahl, Summe : CARDINAL;
```

```
BEGIN
   ReadCard(ErsteZahl);
   ReadCard(ZweiteZahl);
   Summe:=ErsteZahl + ZweiteZahl;
   WriteCard(Summe,5)
END Addition.
```

Probelauf:

7
14
21

3.2 Die Zuweisung

Hier ist wieder etwas Neues: Zu der einzig bisher bekannten Anweisung (Prozeduraufruf) ist eine weitere gekommen: die Zuweisung Summe:=ErsteZahl + ZweiteZahl. Die Zuweisung ist die fundamentalste aller Anweisungen. Sie ermöglicht die Veränderung von Variablen. Ihre allgemeine Form ist:

Zuweisung::= VariablenName ":=" Ausdruck.

Der Begriff «Ausdruck» ist etwas komplizierter zu entschlüsseln. Eine erste Annäherung kann so lauten: Ein Ausdruck besteht aus Operanden und Operatoren. Operanden sind Konstante und Variable. Operatoren sind Verknüpfungszeichen und verbinden zwei Operanden.

Die zugelassenen Operatoren hängen vom Typ der Operanden ab. Bei CARDINAL-Zahlen sind das bsw. die Rechenzeichen + und – oder das Vergleichszeichen < (kleiner). Das Ergebnis einer Operation ist wieder von einem bestimmten Typ.

Wir wollen uns die genaue Definition von «Ausdruck» für den zweiten Teil des Buches aufheben und an dieser Stelle auf unsere Intuition vertrauen, syntaktisch korrekte Ausdrücke zu bilden. Nicht korrekt sind sicherlich Formulierungen wie «1+–9» oder «ErsteZahl –– ZweiteZahl».

3.3 Natürliche Zahlen: CARDINAL

Folgende Rechenoperationen stehen für CARDINAL-Zahlen zur Verfügung:

Zeichen	Bedeutung	Beispiel	Ergebnis
+	Addition	3+19	22
−	Subtraktion	100−97	3
*	Multiplikation	12*9	108
DIV	Ganzzahlige Division	19 DIV 4	4 (Rest bleibt unberücksichtigt)
MOD	Rest bei Division	19 MOD 4	3

Hinweis: «DIV» und «MOD» sind Schlüsselwörter.

Bei der Berechnung eines Ausdrucks (durch den Computer) gilt die Regel, daß Punktrechnungen (*, DIV und MOD) vor Strichrechnungen (+ und −) ausgeführt werden. Zum Ändern der Berechnungsreihenfolge (Priorität) können beliebig viele runde Klammern verwendet werden. Ausdrücke dürfen über mehrere Zeilen reichen. Bei der Ausführung der Zuweisung wird immer erst der Ausdruck vollständig berechnet und schließlich das Ergebnis der Variablen auf der linken Seite des Zuweisungszeichens «:=» zugewiesen.

Noch ein Beispiel: Es soll ein Programm geschrieben werden, das zwei natürliche Zahlen einliest. Ausgegeben werden soll der ganzzahlige Quotient und der Rest der Teilung der ersten durch die zweite Zahl.

Lösung 1:

```
MODULE DivisionMitRest1;
FROM InOut IMPORT ReadCard, WriteCard;
VAR ErsteZahl, ZweiteZahl, Quotient, Rest : CARDINAL;
BEGIN
  ReadCard(ErsteZahl);
  ReadCard(ZweiteZahl);
  Quotient:=ErsteZahl DIV ZweiteZahl;
  Rest:=ErsteZahl MOD ZweiteZahl;
  WriteCard(Quotient,5);
  WriteCard(Rest,5)
END DivisionMitRest1.
```

Probelauf:

```
19
5
    3  4
```

Ausdrücke können auch als Parameter von Prozeduren verwendet werden, wenn diese nur einen Wert benötigen. Ein Beispiel hierfür ist «WriteCard», wo der Wert der übergebenen Zahl auf dem Bildschirm ausgegeben wird. Es sind also auch Aufrufe wie «WriteCard(12+25,5)» oder «WriteCard(19 MOD 4,5)» möglich. Der erste entspricht «WriteCard(37,5)», der letzte «WriteCard(3,5)». Somit kann das obige Programm auch so gelöst werden:

Lösung 2:

```
MODULE DivisionMitRest2;
FROM InOut IMPORT ReadCard, WriteCard;
VAR ErsteZahl, ZweiteZahl : CARDINAL;
BEGIN
  ReadCard(ErsteZahl);
  ReadCard(ZweiteZahl);
  WriteCard(ErsteZahl DIV ZweiteZahl,5);
  WriteCard(ErsteZahl MOD ZweiteZahl,5)
END DivisionMitRest2.
```

Zu allen bisherigen Programmen ist zu bemerken: Sie sind nicht benutzerfreundlich. Nach dem Start steht der Cursor blinkend auf dem Bildschirm, es erscheinen keine Angaben, was jetzt passiert oder passieren soll. Auch die Ausgaben stehen kommentarlos nebeneinander. Wir erkennen schon hier die Wichtigkeit, bei jeder Ein- und Ausgabe auf die Klarheit der Anwenderführung hinzuarbeiten. Das notwendige Werkzeug haben wir bereits in dem Modul «Hallo» kennengelernt: die Prozeduren «WriteString» und «WriteLn». Damit kann schließlich eine befriedigendere Lösung gefunden werden:

Lösung 3:

```
MODULE DivisionMitRest3;
FROM InOut IMPORT WriteLn, WriteString, ReadCard, WriteCard;
VAR ErsteZahl, ZweiteZahl : CARDINAL;
BEGIN
  WriteLn;
  WriteString("Divisionsprogramm");
  WriteLn;
  WriteString("Bitte geben Sie eine ganze positive Zahl ein: ");
  ReadCard(ErsteZahl);
```

```
  WriteLn;
  WriteString("Und jetzt die zweite Zahl: ");
  ReadCard(ZweiteZahl);
  WriteLn;
  WriteString("Der Quotient ist: ");
  WriteCard(ErsteZahl DIV ZweiteZahl,5);
  WriteLn;
  WriteString("Der Rest ist: ");
  WriteCard(ErsteZahl MOD ZweiteZahl,5);
  WriteLn;
END DivisionMitRest3.
```

Probelauf:

Divisionsprogramm
Bitte geben Sie eine ganze positive Zahl ein: 19
Und jetzt die zweite Zahl: 5
Der Quotient ist: 3
Der Rest ist: 4

Selbstverständlich dürfen Sie mehrere Anweisungen in eine Zeile schreiben. Solange sie inhaltlich zusammengehören, ist auch vom Standpunkt der Programmklarheit nichts dagegen einzuwenden.

Lösung 4:

```
MODULE DivisionMitRest4;
FROM InOut IMPORT WriteLn, WriteString, ReadCard, WriteCard;
VAR ErsteZahl, ZweiteZahl : CARDINAL;
BEGIN
  WriteLn; WriteString("Divisionsprogram"); WriteLn;
  WriteString("Bitte geben Sie eine ganze positive Zahl ein: ");
  ReadCard(ErsteZahl); WriteLn;
  WriteString("Und jetzt die zweite Zahl: "); ReadCard(ZweiteZahl); WriteLn;
  WriteString("Der Quotient ist: "); WriteCard(ErsteZahl DIV ZweiteZahl,5);
  WriteLn;
  WriteString("Der Rest ist: "); WriteCard(ErsteZahl MOD ZweiteZahl,5);
  WriteLn;
END DivisionMitRest4.
```

In Modula-2 können Sie CARDINAL-Zahlen nicht nur in der üblichen Dezimaldarstellung angeben, sondern auch die HEX- und Oktaldarstellung verwenden:

Hexadezimal: 0F32H (muß immer mit einer Ziffer beginnen, H wird nachgestellt)
Oktal: 1234Q (Q wird nachgestellt)

Aufgaben:

1. Schreiben Sie ein Programm, das die Differenz zweier Zahlen berechnet. Testen Sie das Programm mit verschiedenen Zahlenpaaren. Was passiert, wenn die Differenz negativ wird?
2. Schreiben Sie ein Programm, das drei Zahlen miteinander multipliziert. Achten Sie auf die nötigen Bildschirmhinweise.

3.4 Ganze Zahlen: INTEGER

Innerhalb der natürlichen Zahlen kann man keine beliebigen Differenzen bilden. Bei vielen alltäglichen Aufgaben (und nicht nur im Finanzwesen) können jedoch negative Ergebnisse auftreten. Aus diesem Grund gibt es einen weiteren Grundtyp, der die Menge der positiven und negativen ganzen Zahlen umfaßt. Sein Name ist INTEGER.
Auf den Datentyp INTEGER können dieselben Operationen angewandt werden wie auf CARDINAL. Hinzu kommt noch der Operator – als Vorzeichen. Beispiel für Ausdrücke vom Typ INTEGER:

```
(19+8)*2-200
-23000+123*(64 DIV 7)
-(73 MOD 9)*(-3)
```

Vielleicht stellen Sie sich jetzt die Frage, welchen Sinn diese Unterscheidung CARDINAL und INTEGER hat, da ja die natürlichen Zahlen (CARDINAL) ganz offensichtlich ein Teil der ganzen Zahlen (INTEGER) sind und dieser Typ somit vollkommen ausreichen würde.

Die Beantwortung dieser Frage wirft ein neues Licht auf die Arbeit mit Modula-2. Durch die strenge Typenbindung kann sowohl der Compiler als auch das Laufzeitsystem eine Typenüberprüfung vornehmen. Das bedeutet, daß schon beim Übersetzen eines Programms ganz offensichtliche Fehlzuweisungen erkannt und zurückgewiesen werden. Wenn sich während des Programmablaufs illegale Werte ergeben (Differenz zweier CARDINAL-Zahlen mit negativem Ergebnis), wird das Programm mit einer entsprechenden Fehlermeldung abgebrochen. Damit ist es möglich, Programmierfehlern schnell auf die Spur zu kommen. Die strenge Typenbindung hat also den Sinn, zur Programmsicherheit und Fehlerfreiheit beizutragen. Bei kleinen Programmen ist der Vorteil

nicht immer unmittelbar ersichtlich, bei großen Programmen hingegen ist die Typenbindung eine unverzichtbare Hilfe.

Prinzipiell können wir hier eine Programmierregel formulieren: Verwenden Sie stets den knappesten Typ für die jeweilige Aufgabe! Wenn es beispielsweise darum geht, irgendwelche Dinge mittels eines Programms zu zählen, so wäre der Bereich der ganzen Zahlen zu groß. Es gibt keine negativen Anzahlen. Somit kann der (knappere) Typ CARDINAL eingesetzt werden. Das Befolgen der obigen Regel bewirkt eine direkte Programmierunterstützung durch das Modula-2-System.

Nur der Vollständigkeit halber sei noch ein weiterer Unterschied angeführt. Für die computerinterne Darstellung von Zahlen wird immer eine bestimmte Anzahl von Speichereinheiten (Speicherworten, engl. words) benötigt. Aus diesem Grund ist der Bereich der Zahlen (CARDINAL oder INTEGER) niemals unendlich groß, es findet immer nur ein Ausschnitt aus den jeweiligen Zahlenmengen Verwendung. Die Größe des Bereiches ist implementationsabhängig. Auf Rechnern mit 32-Bit-Architektur ist er üblicherweise größer als auf solchen, die mit einem 8- oder 16-Bit-Prozessor arbeiten. Es gibt also immer eine größte und eine kleinste INTEGER- und CARDINAL-Zahl. Da der zur Verfügung gestellte Speicherplatz für Variable beider Typen gleich groß ist, ist die größte CARDINAL-Zahl normalerweise immer das doppelte der größten INTEGER-Zahl. Um davon eine Vorstellung zu geben, hier die Werte für 16-Bit-Systeme:

Typ	kleinster Wert	größter Wert
CARDINAL	0	65535
INTEGER	–32768	32767

Die Ein- und Ausgabeprozeduren für INTEGER-Zahlen sind ebenfalls im schon bekannten Modul «InOut» zu finden. Hier nun das Programm, das die Differenz zweier ganzer Zahlen berechnet:

```
MODULE Differenz;
  FROM InOut IMPORT WriteString, WriteLn, ReadInt, WriteInt;
  VAR ErsteZahl, ZweiteZahl : INTEGER;
  BEGIN
    WriteLn; WriteString("Subtraktionprogramm"); WriteLn;
    WriteString("Bitte geben Sie eine ganze Zahl ein: ");
    ReadInt(ErsteZahl); WriteLn;
    WriteString("Und jetzt die zweite Zahl: "); ReadInt(ZweiteZahl); WriteLn;
    WriteString("Die Differenz ist: "); WriteInt(ErsteZahl-ZweiteZahl,5);
    WriteLn
  END Differenz.
```

Probelauf:

Subtraktionsprogramm
Bitte geben Sie eine ganze Zahl ein: 123
Und jetzt die zweite Zahl: 234
Die Differenz ist: -111

Hinweis: In Ausdrücken dürfen INTEGER- und CARDINAL-Zahlen nicht gemischt verwendet werden!

Aufgaben:

1. Schreiben Sie das Divisionsprogramm mit ganzen Zahlen.
2. Testen Sie, wie Ihr Modula-2-System auf Typenmischung reagiert.

3.5 Konvertierung von CARDINAL nach INTEGER und umgekehrt

Werden beide Typen in einem Ausdruck benötigt, so müssen (und können) die Typen umgewandelt werden, je nachdem, welchen Typ der Ausdruck haben soll. Die Vorgehensweise ist sehr einfach.

Zunächst wird der Name des neuen Typs angegeben, gefolgt vom – in runden Klammern eingeschlossenen – Namen der Variablen. Dieses «Umtypisieren» (engl. re-typing) ist mit allen Typen möglich (aber nur manchmal sinnvoll).

Beispiel:

```
VAR  ErsteZahl, ZweiteZahl : CARDINAL;
     Differenz : INTEGER;

     ... Differenz:=INTEGER(ErsteZahl)-INTEGER(ZweiteZahl)

     oder

     ... ErsteZahl:=CARDINAL(Differenz)+ZweiteZahl
```

Hinweis: Die Umtypisierung ist keine echte Typumwandlung. Jede Variable (oder der Wert eines Ausdrucks) wird intern als Bitmuster gespeichert. Wie schon bemerkt, bewirkt die Typangabe eine entsprechende Interpretation dieses Bitmusters. Bei der Umtypisierung findet lediglich eine andere Interpretation dieses Bitmusters statt. Sie ist nur sinnvoll, wenn die Bitmuster der verschiedenen

Typen in Teilbereichen äquivalent sind und zudem der benötigte Speicherplatz gleich groß ist. Das ist bei den Typen CARDINAL und INTEGER im Bereich von 0 bis zur größten INTEGER-Zahl der Fall. Außerhalb dieses Bereichs tritt immer eine Fehlinterpretation auf. Sie können das sichtbar machen, indem Sie mit dem folgenden Programm etwas experimentieren.

```
MODULE ReTypingTest;
  FROM InOut IMPORT ReadCard, ReadInt, WriteCard, WriteInt,
                    WriteString, WriteLn;
  VAR  CardinalZahl : CARDINAL;
       IntegerZahl : INTEGER;
  BEGIN
    WriteLn; WriteString("Typumwandlung mit Umtypisierung"); WriteLn;
    WriteString("Geben Sie eine CARDINAL-Zahl ein: ");
    ReadCard(CardinalZahl); WriteLn;
    WriteString("Geben Sie eine INTEGER-Zahl ein: ");
    ReadInt(IntegerZahl); WriteLn;
    WriteString("Die CARDINAL-Zahl als INTEGER: ");
    WriteInt(INTEGER(CardinalZahl); WriteLn;
    WriteString("Die INTEGER-Zahl als CARDINAL: ");
    WriteCard(CARDINAL(IntegerZahl); WriteLn
  END ReTypingTest.
```

Probelauf:

Typumwandlung mit Umtypisierung
Geben Sie eine CARDINAL-Zahl ein: 1000
Geben Sie eine INTEGER-Zahl ein: -1
Die CARDINAL-Zahl als INTEGER: 1000
Die INTEGER-Zahl als CARDINAL: 65535

3.6 Zeichen: CHAR

Sprachliche Äußerungen, die auf irgendeine Weise niedergelegt werden, nennt man Texte. Sie sind das häufigste mittelbare Kommunikationsmittel für Menschen (im Gegensatz zur unmittelbaren Anrede). Auch dieses Buch ist nichts anderes als ein auf Papier festgehaltener Text. Ein Text besteht aus Buchstaben, Ziffern und Sonderzeichen wie Punkt oder Gänsefüßchen. Wenn Sie einen Text niederschreiben (z. B. einen Programmtext eingeben), so erfahren Sie, daß noch einige weitere Zeichen benötigt werden, um auch die Struktur (Zeilenende, neue Seite, Tabulator) eines Textes festzulegen. Solche Zeichen, die nur der Strukturierung oder Übertragung eines Textes dienen, werden «Steuerzeichen» genannt. Jene, die wirklich sichtbar sind, heißen «druckbare Zeichen». Übrigens

gehört auch das Leerzeichen zu den druckbaren Zeichen, da es indirekt sichtbar ist. Sein Fehlen würde zu einem Zusammenkleben der Wörter führen.

Um einem Wildwuchs bei den Zeichen, die von Computern verarbeitet werden können, vorzubeugen, wurde schon sehr früh ein Zeichenvorrat definiert, der auf allen Rechnern verfügbar sein muß. Der international standardisierte Zeichensatz (ISO-Standard, ISO = International Standards Organisation) heißt in den Vereinigten Staaten ASCII (American Standard Code for Information Interchange, sprich «Aski») und wird auch bei uns so genannt. Er besteht aus 128 Zeichen, von denen die ersten 32 die Steuerfunktionen übernehmen.

	0	20	20	60	100	120	140	160
0	nul	dle		0	@	P	‘	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	’	7	G	W	g	w
10	bs	can	(	8	H	X	h	x
11	ht	em	)	9	I	Y	i	y
12	lf	sub	*	:	J	Z	j	z
13	vt	esc	+	;	K	[	k	{
14	ff	fs	,	<	L	\	l	¦
15	cr	gs	–	=	M	]	m	}
16	so	rs	.	>	N	î	n	~
17	si	us	/	?	O	←	o	del

Leider wurden bei den druckbaren Zeichen nur jene berücksichtigt, die vom englischen Alphabet benötigt werden. So gibt es in der ASCII-Tabelle beispielsweise keine deutschen Umlaute. Aus diesem Grund wurden einige länderspezifische Abwandlungen der ASCII-Tabelle geschaffen. Im deutschen Sprachraum gibt es eine Variante, bei der die Umlaute an die Stelle der Zeichen "[" (Ä), "\" (Ö), "]" (Ü), "{" (ä), "|" (ö), "}" (ü) und " ~" (ß) gesetzt wurden. Einen anderen Ausweg bietet die Ausweitung der Tabelle auf 256 Zeichen. Hier findet vermehrt der Zeichensatz Verwendung, den IBM für seine Personalcomputer geschaffen hat. Diese Erweiterung hat den Vorteil, daß die ersten 128 Zeichen unberührt bleiben.

Als Grundlage für jede sprachliche Kommunikation stellt Modula-2 mit «CHAR» einen Datentyp zur Verfügung, der alle

Zeichen der (u.U. erweiterten) ASCII-Tabelle erfaßt. In «InOut» befinden sich die Prozeduren «Read» und «Write» zur Ein- und Ausgabe genau eines Zeichens. Ein kleines, triviales Programm soll ein Zeichen einlesen und wieder ausgeben:

```
MODULE ZeichenEinAus;
  FROM InOut IMPORT Read, Write;
  VAR Zeichen : CHAR;
  BEGIN
    Read(Zeichen); Write(Zeichen)
  END ZeichenEinAus.
```

Konstante von druckbaren Zeichen werden, wie String-Konstante, in einfache oder doppelte Anführungszeichen eingeschlossen, z. B. 'A' oder "." oder ' ' (Leerzeichen). Steuerzeichen werden durch ihren ASCII-Code angegeben, dem der Buchstabe C direkt nachgestellt wird. Leider muß die Angabe des Codes im Oktalsystem erfolgen und nicht in der gebräuchlichen Dezimaldarstellung. Aber anhand der obigen Tabelle ist der entsprechende Oktalwert leicht zu finden. Am Anfang einer Tabellenspalte stehen die ersten drei Oktalziffern, zu der Sie die Zeilennummer eines bestimmten Zeichens addieren müssen. Beispiel: das Zeichen A (dez. 65) steht in der Spalte mit der Ziffernfolge 100, die Zeilennummer ist 1, also ergibt die Oktaldarstellung 101 und die Zeichenkonstante 101C. Bei der Oktaldarstellung dürfen führende Nullen weggelassen werden.

Beispiele:

Write(007C) oder Write(7C)	läßt den Rechner piepsen
Write(101C)	bringt A auf den Bildschirm

Es dürfte klar sein, daß Daten vom Typ CHAR intern als Zahlen laut ASCII-Tabelle behandelt werden. Somit müßte eine Umwandlung von einem Zeichen in seine entsprechende ASCII-Darstellung und umgekehrt eigentlich leicht möglich sein. Eine Umwandlung durch Umtypisierung scheidet jedoch aus, da der benötigte Speicherplatz von CARDINAL-Zahlen (ein Speicherwort) und Zeichen (ein Byte = ein Bruchteil eines Speicherwortes) beträgt. Aus diesem Grund gibt es die zwei Standardfunktionen «ORD» und «CHR», die diese Umwandlung bewerkstelligen.

Funktionen können wir an dieser Stelle als besondere Operatoren auffassen. Dabei wird der Funktionsname vorangestellt, die Operanden folgen in runden Klammern eingeschlossen und durch

Komma getrennt (wie die Parameter von Prozeduren). Bezieht sich die Funktion nur auf einen Parameter (Argument), so steht selbstverständlich kein Komma. Wie normale Operatoren, erhalten auch Funktionen immer ein Ergebnis (eines ganz bestimmten Typs). Somit können sie auch beliebig in Ausdrücken verwendet werden.

Funktion	Argument	Ergebnis	Bemerkung
ORD	CHAR	CARDINAL	ASCII-Code eines Zeichens
CHR	CARDINAL	CHAR	Zeichen lt. ASCII-Tabelle

Beispiel:

ORD('A') ergibt die CARDINAL-Zahl 65
CHR(65) ergibt das Zeichen 'A'

Wie Sie aus der ASCII-Tabelle ersehen können, haben die Kleinbuchstaben einen um den Wert 32 höheren ASCII-Code als die entsprechenden Großbuchstaben. Ein Programm, das einen Großbuchstaben einliest und diesen kleingeschrieben wieder ausgibt, kann so realisiert werden:

```
MODULE GrossUndKlein;
  FROM InOut IMPORT Read, Write, WriteString, WriteLn;
  VAR Grossbuchstabe, Kleinbuchstabe : CHAR;
      GrossbuchstabenCode, KleinbuchstabenCode : CARDINAL;
  BEGIN
    WriteLn; WriteString("Bitte geben Sie einen Großbuchstaben ein: ")
    Read(Grossbuchstabe);
    GrossbuchstabenCode:=ORD(Grossbuchstabe);
    KleinbuchstabenCode:=GrossbuchstabenCode+32;
    Kleinbuchstabe:=CHR(KleinbuchstabenCode);
    WriteString(" - > "; Write(Kleinbuchstabe); WriteLn
  END GrossUndKlein.
```

Hinweis: Da die Umwandlung von Klein- in Großbuchstaben sehr häufig benötigt wird, gibt es diese als Standardfunktion «CAP».

CAP('a') ergibt A
CAP('z') ergibt Z
CAP('!') ergibt !

(Alle Zeichen außer den Kleinbuchstaben werden nicht umgewandelt.)

Aufgaben:

1. Schreiben Sie das Programm «GrossUndKlein» so um, daß möglichst wenige Variablen benötigt werden. Vergleichen Sie die Programme, und beurteilen Sie die Klarheit der einzelnen Versionen.
2. Schreiben Sie ein Programm, das eine Dezimalzahl (bis max. 127) einliest und diese in Oktaldarstellung ausgibt (ohne die Prozedur «WriteOct» aus InOut zu verwenden). Benützen Sie ausschließlich bisher bekannte Konstruktionen. Geben Sie die Oktalzahl dreistellig mit führenden Nullen aus. Dazu ein Hinweis (für mathematisch Unbedarfte): Die erste Ziffer ist das Ergebnis der ganzzahligen Teilung der Dezimalzahl durch 64. Die zweite Ziffer erhalten Sie, wenn Sie den Rest dieser Teilung durch acht teilen. Und die dritte Ziffer schließlich ist der Rest der zweiten Teilung.

3.7 Wahrheitswerte: BOOLEAN

Die Wahrheit eines Computers hat nichts mit dem philosophischen Begriff «Wahrheit» zu tun. Sie gibt nur schlicht darüber Auskunft, ob ein Sachverhalt, der innerhalb der beschränkten Zahlenwelt eines Rechners darstellbar ist, zutrifft oder nicht. Trifft ein Sachverhalt zu, so wird der Satz, mit dem dieser Sachverhalt ausgedrückt wird, als «wahr» (engl. true) bezeichnet, andernfalls als «falsch» (engl. false). Die Sachverhalte sind meist einfache Vergleiche, wie «Die Zahl A ist größer als die Zahl B» oder Zustände wie «Auf der Tastatur wurde eine Taste gedrückt».

Der Datentyp BOOLEAN hat somit einen sehr kleinen Wertebereich: FALSE und TRUE. Eine Variable dieses Typs kann immer nur einen der beiden Werte annehmen. Trotzdem ist dieser Datentyp für die Programmierung von allerhöchster Wichtigkeit, denn er bildet die Grundlage für die interne Steuerung eines Programms. Je nachdem, ob das Ergebnis eines Ausdrucks vom Typ BOOLEAN den Wert TRUE oder FALSE annimmt, wird das Programm anders weiterarbeiten. Oder eine bestimmte Anweisungsfolge wird so lange wiederholt, bis das Ergebnis eines booleschen Ausdrucks TRUE ist. Aufgrund dieser großen Wichtigkeit werden wir uns an dieser Stelle ausgiebig mit dem Typ BOOLEAN beschäftigen.

Leider gibt es im Modula-2-System keine Ein- und Ausgabeprozeduren für den Datentyp BOOLEAN, weil diese in der normalen

Programmierarbeit nicht benötigt werden. Während des Erlernens der Sprache können sie jedoch viel zur Klärung und Überprüfung beitragen. Deshalb wollen wir an dieser Stelle – unter Vorgriff auf die nächsten Kapitel – ein eigenes Bibliotheksmodul für den erwähnten Zweck entwickeln. Die Erklärung all dessen, was jetzt noch fremd und unverständlich ist, wird etwas später ausführlich nachgeholt.

3.7.1 Das Bibliotheksmodul «BooleanInOut»

Bitte führen Sie Schritt für Schritt diese Arbeitsanleitung aus:

1. Tippen Sie den folgenden Modul-Text ab und lassen ihn von Ihrem Compiler übersetzen.

```
DEFINITION MODULE BooleanInOut;

  PROCEDURE ReadBoolean(VAR B : BOOLEAN);
  PROCEDURE WriteBoolean(B : BOOLEAN);

END BooleanInOut.
```

Falls Sie eine Fehlermeldung erhalten, so fügen Sie nach dem Modul-Kopf folgende Zeile ein:

EXPORT QUALIFIED ReadBoolean, WriteBoolean;

2. Verfahren Sie ebenso mit dem folgenden Text.

```
IMPLEMENTATION MODULE BooleanInOut;

  FROM InOut IMPORT ReadString, WriteString, WriteLn;

  PROCEDURE ReadBoolean(VAR B : BOOLEAN);
    VAR Eingabe : ARRAY[0..10] OF CHAR;
        Okay : BOOLEAN;

    PROCEDURE gleich(X : ARRAY OF CHAR; Y : ARRAY OF CHAR) : BOOLEAN;
      VAR i : CARDINAL;

      PROCEDURE min(X,Y : CARDINAL) : CARDINAL;
       BEGIN
         IF X<Y THEN RETURN X ELSE RETURN Y END
       END min;

      BEGIN (* gleich *)
        FOR i:=0 TO min(HI(X),HI(Y)) DO
          IF X[i]<>Y[i] THEN RETURN FALSE END
        END; (* FOR *)
        RETURN TRUE
      END gleich;
```

```
    BEGIN (* ReadBoolean *)
      Okay:=FALSE;
      REPEAT
        ReadString(Eingabe);
        IF NOT(gleich(Eingabe,'TRUE') OR gleich(Eingabe,'FALSE'))
          THEN WriteString(" - ?"); WriteLn
          ELSE OKay:=TRUE
          END (* IF *)
       UNTIL Okay;
       B:=gleich(Eingabe,'TRUE')
    END ReadBoolean;

  PROCEDURE WriteBoolean;
    BEGIN
      IF B THEN WriteString("TRUE") ELSE WriteString("FALSE") END
    END WriteBoolean;

END BooleanInOut.
```

Jetzt steht Ihnen ein weiteres Bibliotheksmodul zur Verfügung, aus dem Sie die Prozeduren «ReadBoolean» und «WriteBoolean» importieren können. Ein kleines Programm zum Testen dieses Moduls:

```
MODULE BooleanTest;
  FROM BooleanInOut IMPORT ReadBoolean, WriteBoolean;
  FROM InOut IMPORT WriteString, WriteLn;
  VAR Eingabe : BOOLEAN;
  BEGIN
    WriteLn; WriteString("Bitte geben Sie 'TRUE' oder 'FALSE' ein: ");
    ReadBoolean(Eingabe); WriteLn;
    WriteString("Es wurde ");
    WriteBoolean(Eingabe);
    WriteString(" eingegeben."); WriteLn
  END BooleanTest.
```

3.7.2 Operationen mit booleschem Ergebnis

Zur Formulierung einfacher Aussagen stehen die sog. «relationalen Operatoren» zur Verfügung. Damit können typengleiche Ausdrücke miteinander verglichen werden. Je nachdem, ob der ausgedrückte Vergleich zutrifft oder nicht, erhält die Aussage den Wert TRUE oder FALSE.

Operator	Bedeutung
=	gleich
<> oder #	ungleich
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich

Diese Operatoren sind auf alle Grundtypen anwendbar. Während der Einsatz bei Zahlen klar ist (von zwei Zahlen kann immer bestimmt werden, welche die größere ist oder ob sie gleich sind), ist der Vergleich von Zeichen (CHAR) und Wahrheitswerten (BOOLEAN) auf «größer» und «kleiner» nicht unmittelbar einleuchtend. Nun, den Zeichen liegt ganz einfach die ASCII-Tabelle zugrunde. Jedem Zeichen entspricht eine Zahl, und diese Zahlen können ganz normal verglichen werden.

Beispiele:

'A' < 'B' ergibt TRUE
'Z' <= '0' ergibt FALSE
'.' >= '!' ergibt FALSE

Die Wahrheitswerte sind willkürlich angeordnet. Bei der Definition von Modula-2 wurde festgelegt: FALSE < TRUE. Die Frage, ob es überhaupt sinnvoll ist, Wahrheitswerte größenmäßig zu vergleichen, können wir erst weiter unten beantworten.

Hier ein kleines Programm zur Demonstration der relationalen Operatoren:

```
MODULE Vergleiche;
  FROM BooleanInOut IMPORT WriteBoolean;
  FROM InOut IMPORT WriteString, WriteReal, ReadCard, WriteCard;
  VAR x, y : CARDINAL;
  BEGIN
    WriteString("Die relationalen Operatoren"); WriteLn;
    WriteString("---------------------------"); WriteLn;
    WriteLn;
    WriteString("Bitte geben Sie zwei CARDINAL-Zahlen ein:"); WriteLn;
    WriteString("x = "); ReadCard(x); WriteLn;
    WriteString("y = "); ReadCard(y); WriteLn;
    WriteLn;
    WriteString("x <  y   -> "); WriteBoolean(x<y); WriteLn;
    WriteString("x <= y   -> "); WriteBoolean(x<=y); WriteLn;
    WriteString("x =  y   -> "); WriteBoolean(x =y); WriteLn;
    WriteString("x >  y   -> "); WriteBoolean(x>y); WriteLn;
    WriteString("x >= y   -> "); WriteBoolean(x>=y); WriteLn
  END Vergleiche.
```

Probelauf:

```
Die relationalen Operatoren
---------------------------
Bitte geben Sie zwei CARDINAL-Zahlen ein:
x = 1250
y = 333
```

```
x <  y  -> FALSE
x <= y  -> FALSE
x =  y  -> FALSE
x >  y  -> TRUE
x >= y  -> TRUE
```

3.7.3 Funktionen mit booleschem Ergebnis

In Modula-2 gibt es genau eine Standardfunktion, die ein Ergebnis vom Typ BOOLEAN liefert: ODD(x) gibt an, ob die (CARDINAL- oder INTEGER-)Zahl x ungerade oder gerade (ohne Rest durch zwei teilbar) ist.

```
ODD(13)   ergibt TRUE
ODD(-14)  ergibt FALSE
```

Später werden wir die Möglichkeit kennenlernen, selbst Funktionen mit booleschem Ergebnis zu schreiben.

3.7.4 Junktoren

Vergleiche, boolesche Variable, Funktionen und Konstante sind die elementaren Bausteine (oder Aussagen). Mit den Bindewörtern (Junktoren) «NOT» (nicht), «AND» (und) und «OR» (oder) können elementare Aussagen zu komplexen Sätzen verknüpft werden. Dabei hängt der Wahrheitswert der komplexen Aussage ausschließlich von den Wahrheitswerten der Einzelsätze und dem verwendeten Junktor ab.

«NOT» (Negation) dreht den Wahrheitswert um. «NOT TRUE» ergibt «FALSE» und «NOT FALSE» ergibt «TRUE». Somit ist klar, daß eine doppelte Negation wieder den Ausgangswert liefert: «NOT (NOT TRUE)» ergibt «NOT FALSE», und das wiederum ist gleichbedeutend mit «TRUE». «NOT» hat für Wahrheitswerte offensichtlich eine ähnliche Funktion wie das Vorzeichen – bei ganzen Zahlen.

«AND» (Konjunktion) entspricht dem umgangssprachlichen «und». Werden zwei Aussagen mit «AND» verbunden, so wird die zusammengesetzte Aussage nur dann «TRUE», wenn beide Teile den Wert «TRUE» haben, ansonsten ist sie immer «FALSE».

«OR» (Adjunktion) übernimmt die Funktion des «nichtausschließenden oder». «SATZ1 OR SATZ2» bedeutet also «SATZ1

ODER SATZ2 ODER BEIDE», und nicht «ENTWEDER SATZ1 ODER SATZ2». Eine mit «OR» verknüpfte Aussage wird nur dann «FALSE», wenn beide Komponenten «FALSE» sind, andernfalls wird sie «TRUE».

Die Funktion der Junktoren kann in einer sog. Wahrheitstabelle dargestellt werden:

Satz1	Satz2	NOT Satz1	Satz1 AND Satz2	Satz1 OR Satz2
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
TRUE	TRUE	FALSE	TRUE	TRUE

Werden boolesche Ausdrücke mit Hilfe der angesprochenen Junktoren gebildet, sind folgende Prioritätsfolgen zu beachten:

- ☐ NOT bindet stärker als AND.
- ☐ AND bindet stärker als OR.
- ☐ Junktoren binden stärker als relationale Operatoren.
- ☐ Zum Verändern der Priorität können runde Klammern verwendet werden.

3.7.5 Abarbeitung boolescher Ausdrücke in Modula-2

In Ausdrücken werden die einzelnen Operatoren ihrer Priorität entsprechend bearbeitet, solche gleicher Priorität von links nach rechts. Geklammerte Ausdrücke werden nach dieser Regel von innen nach außen berechnet.

Bei Ausdrücken, die mit den Junktoren «UND» und «OR» gebildet werden, ist jedoch eine Besonderheit zu beachten. Da ein Satz, der aus zwei (oder mehreren) Teilsätzen mit «AND» gebildet wird, auf jeden Fall «FALSE» ergibt, wenn ein Teilsatz «FALSE» ist, wird beim ersten Auftreten von «FALSE» hier die Berechnung abgebrochen (und für die übrigen Teilsätze nicht mehr ausgeführt). Entsprechendes gilt für «OR», nur daß hier ein erstes Auftreten von «TRUE» zum Abbruch führt. Modula-2 führt also überflüssige Berechnungen nicht durch!

Beispiel:

```
VAR  IstImBereich : BOOLEAN;
     Zahl : CARDINAL;
...
IstImBereich:=(Zahl>=5) AND (Zahl<=10);
```

Hat die Variable Zahl z. B. den Wert 0, so wird die Operation «(Zahl<=10)» nicht durchgeführt, da schon der Vergleich «(Zahl>=5)» den Wert «FALSE» ergibt.

```
MODULE JunktorenLogik;
  FROM BooleanInOut IMPORT ReadBoolean, WriteBoolean;
  FROM InOut IMPORT WriteString, WriteLn;
  VAR Satz1, Satz2 : BOOLEAN;
  BEGIN
    WriteString("Demonstration der Junktoren"); WriteLn;
    WriteString("---------------------------"); WriteLn;
    WriteLn;
    WriteString("Geben Sie bitte zwei Wahrheitswerte ein:"); WriteLn;
    ReadBoolean(Satz1); WriteLn;
    ReadBoolean(Satz2); WriteLn;
    WriteLn;
    WriteString("Satz1: "); WriteBoolean(Satz1); WriteLn;
    WriteString("Satz2: "); WriteBoolean(Satz2); WriteLn;
    WriteLn;
    WriteString("NOT Satz1        <-> ");
    WriteBoolean(NOT Satz1); WriteLn;
    WriteString("NOT Satz2        <-> ");
    WriteBoolean(NOT Satz2); WriteLn;
    WriteString("Satz1 AND Satz2  <-> ");
    WriteBoolean(Satz1 AND Satz2); WriteLn;
    WriteString("Satz1 OR  Satz2  <-> ");
    WriteBoolean(Satz1 OR  Satz2); WriteLn
  END JunktorenLogik.
```

Probelauf:

```
Demonstration der Junktoren
---------------------------
Geben Sie bitte zwei Wahrheitswerte ein:
FALSE
TRUE

Satz1: FALSE
Satz2: TRUE

NOT Satz1        <-> TRUE
NOT Satz2        <-> FALSE
Satz1 AND Satz2  <-> FALSE
Satz1 OR Satz2   <-> TRUE
```

3.7.6 Komplexe Sätze

Mit den Junktoren können weitere logische Verknüpfungen realisiert werden. Als erste wäre hier «entweder ... oder» zu nennen. Umgangssprachlich könnte man die Beziehung «Entweder Satz1 oder Satz2» so darstellen: «Satz1 oder Satz2, aber (= und) nicht Satz1 und Satz2». Übertragen in die Junktoren-Sprache ergibt sich:

(Satz1 OR Satz2) AND NOT(Satz1 AND Satz2)

Wir wollen die Wahrheitswerte dieses Satzes in einer Tabelle darstellen:

Satz1	Satz2	(Satz1 OR Satz2)	NOT(Satz1 AND Satz2)	... AND ...
FALSE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	FALSE	FALSE

Wir sehen, daß die zusammengesetzte Aussage immer nur dann «TRUE» wird, wenn genau eine der beiden Teilaussagen «TRUE» ist. Weiterhin sehen wir, daß immer dann, wenn «Satz1» denselben Wert wie «Satz2» hat, wenn also «Satz1=Satz2» ist, das Ergebnis der Gesamtaussage «FALSE» ist. Demnach können wir die «entweder ... oder»-Beziehung wesentlich einfacher ausdrücken: «Satz1 <>Satz2».

Haben zwei boolesche Ausdrücke dieselbe Wahrheitstabelle, so sagt man, daß die Ausdrücke «logisch äquivalent» (gleichbedeutend) sind. Für die Programmierarbeit ist wichtig, daß jeder Ausdruck durch einen logisch äquivalenten Ausdruck ersetzt werden kann. Oft kann man einen komplizierten Ausdruck durch einen wesentlich einfacheren ersetzen, was einerseits zu einer schnelleren Programmausführung führt, andererseits die Programme klarer und leichter nachvollziehbar macht.

Hier nun eine Liste der wichtigsten Äquivalenzen (A und B sind boolesche Ausdrücke):

Ausdruck	äquival. Ausdruck	Bedeutung
(A OR B) AND NOT (A AND B)	A<>B (oder A # B)	entweder A oder B
NOT A OR B	A<=B	A impliziert B, wenn A, dann B
(NOT A OR B) AND (NOT B OR A)	A=B	A äquivalent B, A genau dann, wenn B
NOT (A AND B)	NOT A OR NOT B	nicht (A und B) 'NAND'
NOT (A OR B)	NOT A AND NOT B	nicht (A oder B) 'NOR'

Die letzten beiden Formeln (Gesetze von de Morgan) sind besonders interessant. Sie erlauben die Transformation von «AND» in «OR» und umgekehrt. Im Zusammenhang mit dem verkürzten Abarbeiten von Ausdrücken kann hier der fortgeschrittene Programmierer optimale boolesche Ausdrücke erstellen. Ein kleines Programm soll uns die Gleichwertigkeit der Ausdrücke demonstrieren.

```
MODULE DeMorgan;
  FROM BooleanInOut IMPORT ReadBoolean, WriteBoolean;
  FROM InOut IMPORT WriteString, WriteLn;
  VAR Satz1, Satz2 : BOOLEAN;
  BEGIN
    WriteString("Demonstration der Gesetze von de Morgan"); WriteLn;
    WriteString("---------------------------------------"); WriteLn;
    WriteLn;
    WriteString("Geben Sie bitte zwei Wahrheitswerte ein:"); WriteLn;
    ReadBoolean(Satz1); WriteLn;
    ReadBoolean(Satz2); WriteLn;
    WriteLn;
    WriteString("Satz1: "); WriteBoolean(Satz1); WriteLn;
    WriteString("Satz2: "); WriteBoolean(Satz2); WriteLn;
    WriteLn;
    WriteString("NOT (Satz1 AND Satz2)  <-> ");
    WriteBoolean(NOT (Satz1 AND Satz2)); WriteLn;
    WriteString("NOT Satz1 OR NOT Satz2 <-> ");
    WriteBoolean(NOT Satz1 OR NOT Satz2); WriteLn
  END DeMorgan.
```

Probelauf:

```
Demonstration der Gesetze von de Morgan
---------------------------------------
Geben Sie bitte zwei Wahrheitswerte ein:
```

```
TRUE
FALSE

Satz1: TRUE
Satz2: FALSE

NOT (Satz1 AND Satz2)    <-> TRUE
NOT Satz1 OR NOT Satz2 <-> TRUE
```

3.7.7 Logische Transformationen

Abschließend wollen wir noch einen Blick auf die wichtigsten Transformationen von Vergleichen werfen. Dabei sind x und y Ausdrücke von einem geordneten Typ (CARDINAL, INTEGER, CHAR, BOOLEAN und REAL).

Gleichheit und Ungleichheit:

```
x = y      <->   NOT ( x <> y )
x <> y     <->   NOT ( x = y )
```

Kleiner und größer:

```
x < y      <->   NOT ( x >= y )
x > y      <->   NOT ( x <= y )
x <= y     <->   NOT ( x > y )
x >= y     <->   NOT ( x < y )
```

Regel: Verwenden Sie immer den einfachsten logischen Ausdruck, beispielsweise anstelle von «NOT (x > y)» den gleichwertigen Satz «x <= y».

Hinweis: In Modula-2 sind folgende Abkürzungen erlaubt:

```
~ für NOT
& für AND
```

Aufgaben:

1. Was bewirkt der Prozeduraufruf
   'WriteBoolean(TRUE AND NOT (FALSE OR (TRUE AND NOT FALSE)))'
2. Vereinfachen Sie die Ausdrücke:
   a) (x<y) OR (x=y)

b) NOT (NOT(a<b) AND (c>d))
b) NOT(p AND NOT q) (p und q vom Typ BOOLEAN)

3.8 Fließkommazahlen: REAL

Als letzten Grundtyp stellt Modula-2 die sog. Fließkommazahlen zur Verfügung. Es handelt sich dabei um Dezimalbrüche, die in der wissenschaftlichen Notation mit Mantisse und Exponent geschrieben werden (wie bei einem Taschenrechner). Anstelle des bei uns gebräuchlichen Kommas steht allerdings der (englische) Dezimalpunkt.

Die Syntax von REAL-Zahlen:

REAL-Zahl::=Mantisse[Exponent].
Mantisse::=["+"|"-"]Ziffernfolge"."[Ziffernfolge].
Ziffernfolge::=Ziffer{Ziffer}.
Ziffer::="0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
Exponent::="E"["+"|"-"]Ziffernfolge.

Beispiele für REAL-Zahlen:

1.0 -3.28 +25.E-9 1.24E30

Der Exponent gibt an, um wie viele Stellen der Dezimalpunkt verschoben wird (fehlende Stellen werden mit Nullen aufgefüllt).

3.5E4 = 35000
3.5E-4 = 0.00035
1.0E6 = 1000000
1.0E-10 = 0.0000000001

Hinweis: Der Dezimalpunkt muß grundsätzlich angegeben werden, auch wenn rechts davon keine Ziffer mehr folgt.

Für die Ein- und Ausgabe von Daten des Typs REAL gibt es ein eigenes Bibliotheksmodul namens «RealInOut». Es stellt die Prozeduren «ReadReal» und «WriteReal» zur Verfügung. Mit «ReadReal» wird eine REAL-Variable eingelesen, mit «WriteReal» wird ein REAL-Ausdruck ausgegeben. Bei der Eingabe kann der Dezimalpunkt weggelassen werden. Wie bei «WriteCard» und «WriteInt» muß bei «WriteReal» mit einem zweiten Parameter die

Feldbreite angegeben werden, innerhalb der die REAL-Zahl ausgegeben wird. Mit «WriteReal» wird eine REAL-Zahl immer in der normierten Form ausgegeben. Das bedeutet, daß der Absolutbetrag der Mantisse immer zwischen 1 (eingeschlossen) und 10 (ausgeschlossen) liegt. Einzige Ausnahme ist die Zahl 0.

Beispiele:

Eingabe (ReadReal(x))	Ausgabe (WriteReal(x,20))
0	0.00000000000000E0
1	1.00000000000000E0
10	1.00000000000000E1
100	1.00000000000000E2
0.0005	5.00000000000000E-4

Diese Ausgabeform ist nicht immer befriedigend. Deshalb stellen viele Modula-2-Implementationen noch weitere Ausgabeprozeduren zur Verfügung. Wir werden selbst eine solche Prozedur erstellen.

Als Beispiel wollen wir die Volumenberechnung eines rechteckigen Kastens anhand dessen Länge, Breite und Höhe durchführen:

```
MODULE Volumen;
  FROM RealInOut IMPORT ReadReal, WriteReal;
  FROM InOut IMPORT WriteLn, WriteString;
  VAR Laenge, Breite, Hoehe , Volumen : REAL;
  BEGIN
    WriteString("Volumen eines rechteckigen Kastens"); WriteLn;
    WriteString("----------------------------------"); WriteLn;
    WriteLn;
    WriteString("Länge  (in cm): "); ReadReal(Laenge); WriteLn;
    WriteString("Breite (in cm): "); ReadReal(Breite); WriteLn;
    WriteString("Höhe   (in cm): "); ReadReal(Hoehe); WriteLn;
    WriteLn;
    WriteString("Das Volumen des Kastens (in cm^3): ");
    Volumen:=Laenge*Breite*Hoehe;
    WriteReal(Volumen,15); WriteLn
  END Volumen.
```

Probelauf:

```
Volumen eines rechteckigen Kastens
----------------------------------
Länge     (in cm): 225
Breite    (in cm): 22.8
Höhe      (in cm): 18.3

Das Volumen des Kastens (in cm^3): 9.387899999999998E4
```

Aufgaben:

1. Schreiben Sie die Syntaxdiagramme in EBNF-Notation für CARDINAL- und INTEGER-Zahlen.
2. Schreiben Sie ein Programm, das den Umfang eines rechtwinkeligen Rasenstücks (Länge, Breite) berechnet. Achten Sie auf vernünftige Ein- und Ausgabeanweisungen.

3.9 Zusammenfassung der Grundtypen

Die Typen «CARDINAL», «INTEGER», «CHAR» und «BOOLEAN» haben eins gemeinsam: Sie sind aufzählbar. Das heißt, daß zu jeder gegebenen Größe aus einer dieser Typen (falls vorhanden) der Nachfolger und der Vorgänger bestimmt werden können. Oder anders ausgedrückt: Zwischen zwei Werten eines dieser Typen liegt eine Anzahl von anderen Werten, und diese Anzahl ist völlig unabhängig von der jeweiligen Rechenanlage. So liegen beispielsweise zwischen den CARDINAL-Zahlen 1 und 4 auf jedem Rechner dieser Welt die Zahlen 2 und 3. Das ist bei den REAL-Zahlen nicht der Fall! Die Anzahl der REAL-Zahlen zwischen 0.5 und 0.6 kann ohne Kenntnis der jeweiligen Implementierung nicht beantwortet werden.

Die Unterscheidung in aufzählbare und nichtaufzählbare Typen ist sehr wichtig, da verschiedene Anweisungen ausschließlich mit aufzählbaren Typen arbeiten. Sie werden häufig «einfache Typen» (engl. simple types) oder «skalare Typen» genannt. Jedem Wert eines aufzählbaren Typs entspricht genau eine ganze Zahl (Ordinalwert). Dieser Wert kann mit der Standardfunktion «ORD» bestimmt werden, die wir bereits im Zusammenhang mit dem Datentyp «CHAR» kennengelernt haben.

Hier folgt eine Liste aller Standardprozeduren und -funktionen, die mit allen aufzählbaren Typen einsetzbar sind:

Name	Art	Bedeutung	Beispiel
ORD(x)	Funktion	Ordinalwert von x	ORD('A') (= 65) ORD(FALSE) (= 0)
INC(x)	Prozedur	Nachfolger von x	x:=10; INC(x); (entspricht x:=x+1)
INC(x,n)	Prozedur	n-t Nachfolger von x	x:=10; INC(x,5); (entspricht x:=x+15)

DEC(x)	Prozedur	Vorgänger von x	x:=10; DEC(x); (entspricht x:=x-1)
DEC(x,n)	Prozedur	n-te Vorgänger von x	x:=10; DEC(x,5); (entspricht x:=x-5)
VAL(T,x)	Funktion	Wert des Typs T mit Ordnungszahl x	VAL(BOOLEAN,1) (= TRUE, da ORD(TRUE)=1)

Für alle Grundtypen gibt es die Funktionen «MIN» und «MAX», die als Argument einen Typ-Namen benötigen und den kleinsten bzw. größten Wert dieses Typs als Ergebnis liefern.

MIN(CARDINAL) = 0
MAX(CARDINAL) = 65535
MIN(BOOLEAN) = FALSE usw.

Mit folgendem Programm können Sie den Bereich der REAL-Zahlen ihres Modula-2-Systems bestimmen.

```
MODULE RealBereich;
  FROM RealInOut IMPORT WriteReal;
  FROM InOut IMPORT WriteLn, WriteString;
  BEGIN
   WriteString("Die kleinste REAL-Zahl: ");
   WriteReal(MIN(REAL),20); WriteLn;
   WriteString("Die größte REAL-Zahl: ");
   WriteReal(MAX(REAL),20); WriteLn
  END RealBereich.
```

4 Anweisungen

Bisher haben wir zwei Anweisungen kennengelernt: Prozeduraufruf und Zuweisung. In diesem Kapitel wenden wir uns nun den sog. «strukturierten» Anweisungen zu. Sie heißen deshalb so, weil ihre einzelnen Komponenten wiederum Anweisungen sind.

```
Anweisung::=[Prozeduraufruf | Zuweisung | IF-Anweisung | REPEAT-Anweisung |
             WHILE-Anweisung | FOR-Anweisung | LOOP-Anweisung |
             CASE-Anweisung | RETURN-Anweisung | WITH-Anweisung].
```

Das sind alle Anweisungen in Modula-2. Interessant sind bei dieser Definition die eckigen Klammern. Sie besagen, daß eine Anweisung auch aus keinen Zeichen bestehen kann. Es handelt sich dann um die «leere Anweisung». Vielleicht haben Sie bei Ihren Versuchen schon einmal vor dem abschließenden END ein Semikolon geschrieben. Trotzdem hat der Compiler daran nicht Anstoß genommen. Es wurde ganz einfach noch eine leere Anweisung angenommen und damit die Syntaxregel erfüllt. Die Zulassung der leeren Anweisung ist eine Erleichterung für den Programmierer.

Bis auf die RETURN-Anweisung, die erst bei den Prozeduren behandelt wird, und die WITH-Anweisung, die nur zusammen mit einem neuen Datentyp Verwendung findet, werden die Modula-2-Anweisungen jetzt systematisch vorgestellt.

4.1 IF-Anweisung

Häufig kommt es vor, daß ein Programm anders weiterarbeiten muß, je nachdem, ob eine Berechnung ein bestimmtes Resultat erbrachte oder nicht. In diesem Fall kommt die IF-Anweisung zum Einsatz. Ihre allgemeine Form ist:

```
IF-Anweisung::="IF" BoolescherAusdruck
               "THEN" Anweisungssequenz
               {"ELSIF" BoolescherAusdruck "THEN" Anweisungssequenz}
               ["ELSE" Anweisungssequenz]
               "END".
```

Die booleschen Ausdrücke wurden bereits ausgiebig behandelt. Den Begriff «Anweisungssequenz» wollen wir uns nochmals in Erinnerung rufen:

Anweisungssequenz::=Anweisung {";" Anweisung}.

Hinweis: Eine Anweisungssequenz kann auch nur aus einer leeren Anweisung bestehen, also selbst leer sein.

4.1.1 Die Grundform

Betrachten wir zunächst den einfachsten Fall einer IF-Anweisung, bei dem alle optionalen Teile weggelassen werden:

IF BoolescherAusdruck THEN Anweisungssequenz END

Die Arbeitsweise ist schnell erklärt. Die auf «THEN» folgende Anweisungssequenz wird nur dann abgearbeitet, wenn der boolesche Ausdruck den Wert «TRUE» erhält, andernfalls nicht.

```
MODULE EndeTest;
  FROM InOut IMPORT Read, WriteString, WriteLn;
  VAR Antwort : CHAR;
BEGIN
  WriteLn;
  WriteString("Wenn Sie das Programm abbrechen wollen,"); WriteLn;
  WriteString("geben Sie 'J' ein: ");
  Read(Antwort); WriteLn;
  IF Antwort="J"
  THEN WriteString("Abbruch!"); WriteLn; HALT
  END;
  WriteString("Das Programm wird fortgesetzt.")
END EndeTest.
```

Probelauf:

```
Wenn Sie das Programm abbrechen wollen,
geben Sie 'J' ein: J
Abbruch!

Wenn Sie das Programm abbrechen wollen,
geben Sie 'J' ein: x
Das Programm wird fortgesetzt.
```

In diesem Beispiel findet eine neue Standardprozedur Verwendung: «HALT». Sie macht genau, was ihr Name aussagt. Das Programm wird an dieser Stelle abgebrochen.

Zur eingesetzten IF-Anweisung gibt es folgendes zu bemerken: Bei der Quelltext-Formatierung ist es günstig, die reservierten Wörter (IF, THEN, END) untereinander zu schreiben. Dadurch wird klar, was alles zu dieser Anweisung gehört.

Wenn die Anweisungssequenz sehr umfangreich ist und (oder) wiederum strukturierte Anweisungen enthält, ist bei dem abschließenden «END» oft nicht leicht zu durchschauen, auf welche Anweisung es sich bezieht. Aus diesem Grund wird hier gern die Quelle des «END» in Form eines kurzen Kommentars angegeben: END (* IF *)

Abfragen der obigen Art kommen in der Programmierpraxis sehr oft vor, denn damit ermöglicht man es dem Anwender, mit einfachen Tastendrücken ein Programm zu steuern. Meist soll jedoch das Programm auch die entsprechenden Kleinbuchstaben akzeptieren. Dazu muß nur der boolesche Ausdruck umgewandelt werden:

IF (Antwort="J") OR (Antwort="j") THEN ...

oder

IF CAP(Antwort)="J" THEN ...

4.1.2 Der ELSE-Zweig

Auch die Arbeitsweise der erweiterten IF-Anweisung ist einleuchtend:

IF BoolescherAusdruck
THEN Anweisungssequenz1
ELSE Anweisungssequenz2
END

Ergibt der boolesche Ausdruck den Wert «TRUE», so wird Anweisungssequenz1 ausgeführt, andernfalls Anweisungssequenz2. In einem Beispiel soll geprüft werden, ob eine eingegebene INTEGER-Zahl positiv (>=0) oder negativ ist.

```
MODULE PositivNegativ;
  FROM InOut IMPORT ReadInt, WriteString, WriteLn;
VAR Zahl : INTEGER;
BEGIN
  WriteLn;
  WriteString("Bitte geben Sie eine ganze Zahl ein: ");
  ReadInt(Zahl); WriteLn;
```

```
    IF Zahl>=0
    THEN WriteString("Ihre Zahl ist positiv.")
    ELSE WriteString("Ihre Zahl ist negativ.")
    END;
    WriteLn
END PositivNegativ.
```

Probelauf:

Bitte geben Sie eine ganze Zahl ein: 125
Ihre Zahl ist positiv.

Bitte geben Sie eine ganze Zahl ein: –16
Ihre Zahl ist negativ.

Da die Anweisungssequenz1 dann ausgeführt wird, wenn der boolesche Ausdruck «TRUE» ergibt, die Anweisungssequenz2 dann, wenn das Resultat «FALSE» ist, kann diese Form der bedingten Anweisung ohne Einfluß auf das Programm umgestellt werden:

IF NOT BoolescherAusdruck
THEN Anweisungssequenz2
ELSE Anweisungssequenz1
END

In unserem Beispiel würde diese Transformation zu folgender Anweisung führen:

IF Zahl<0
THEN WriteString("Ihre Zahl ist negativ.")
ELSE WriteString("Ihre Zahl ist positiv.")
END

Dieser Sachverhalt kann manchmal ausgenutzt werden, um die Bedingung (boolescher Ausdruck) einfacher zu gestalten. Wenn Sie etwas fortgeschrittener sind, sollten Sie sich stets der logischen Transformationsregeln erinnern.

4.1.3 Alternativen mit ELSIF

Stehen mehr als zwei Alternativen zur Auswahl, so bedient man sich der «ELSIF»-Zweige:

IF BoolescherAusdruck1
THEN Anweisungssequenz1
ELSIF BoolescherAusdruck2
THEN Anweisungssequenz2
ELSIF BoolescherAusdruck3
THEN Anweisungssequenz3
...
ELSE AlternativeAnweisungssequenz
END

Die Ausführung geschieht folgendermaßen: Erst wird der «BoolescheAusdruck1» berechnet. Ist sein Wert «TRUE», so wird «Anweisungssequenz1» abgearbeitet. Andernfalls wird der «BoolescheAusdruck2» berechnet. Ist dessen Ergebnis «TRUE», so wird «Anweisungssequenz2» ausgeführt. Andernfalls wird das Verfahren für den «BooleschenAusdruck3» wiederholt usw. Nur wenn alle booleschen Ausdrücke den Wert «FALSE» ergeben, wird die Anweisungssequenz nach «ELSE» abgearbeitet.

In einem Beispiel soll das Vorzeichen (Signum) einer Fließkommazahl bestimmt werden. Es gilt

Signum(x)	= +1	falls x>0.0
	−1	falls x<0.0
	0	falls x=0.0

```
MODULE Signum;
  FROM InOut IMPORT WriteString, WriteLn;
  FROM RealInOut IMPORT ReadReal;
  VAR TestZahl : REAL;
  BEGIN
    WriteString("Bestimmung des Signums einer reellen Zahl x"); WriteLn;
    WriteString("-----------------------------------------"); WriteLn;
    WriteLn; WriteString("x = ");
    ReadReal(TestZahl); WriteLn;
    WriteString("Das Signum ist ");
    IF TestZahl>0.0
    THEN WriteString("+1")
    ELSIF TestZahl<0.0
    THEN WriteString("-1")
    ELSE WriteString("0")
    END; (* IF *)
    WriteLn
  END Signum.
```

Probelauf:

```
Bestimmung des Signums einer reellen Zahl x
-------------------------------------------------------------
x = -124.25
Das Signum ist -1
```

Hinweis: Der Einsatz vieler ELSIF-Zweige kann zu unübersichtlichen Programmen führen. Sie sollten hier ganz besonders auf eine saubere optische Gliederung achten!

Betrachten wir noch einmal die Syntax der IF-Anweisung. Hier kommen Anweisungssequenzen vor, also durch Semikolon getrennte Folgen von Anweisungen. Da auch die IF-Anweisung eine Anweisung ist, kann sie hier überall wieder eingesetzt werden. Man sagt auch: Die strukturierten Anweisungen sind rekursiv definiert. Der Begriff der Rekursion wird zwar erst später behandelt. Eine wichtige Konsequenz kann aber schon hier angeführt werden: IF-Anweisungen können beliebig verschachtelt werden.

Aufgaben:

1. Warum ist eine Konstruktion mit «ELSIF» einer mehrfach verschachtelten IF-Anweisung vorzuziehen?
2. Schreiben Sie ein Programm, das Groß- in Kleinbuchstaben umwandelt und umgekehrt.

4.2 Die REPEAT-Anweisung

Modula-2 bietet vier verschiedene Wiederholungsanweisungen. Die einfachste Form lautet etwa folgendermaßen: Wiederhole eine Anweisungssequenz so lange, bis ein boolescher Ausdruck den Wert «TRUE» erhält.

REPEAT-Anweisung::="REPEAT„ Anweisungssequenz "UNTIL" BoolescherAusdruck.

Der boolesche Ausdruck wird immer erst berechnet, wenn die Anweisungssequenz ausgeführt wurde. Demnach wird die Anweisungssequenz mindestens einmal abgearbeitet. Man spricht deshalb auch von einer «nicht-abweisenden Schleife».

Eine wichtige Anwendung als Beispiel. Der Anwender soll eine Frage mit «J» (für Ja) oder «N» (für Nein) beantworten. Dabei soll zwischen Groß- und Kleinbuchstaben nicht unterschieden werden. Alle anderen Eingaben sollen mit einer erneuten Anforderung zurückgewiesen werden.

```
MODULE JaNein1;
  FROM InOut IMPORT Read, WriteString, WriteLn;
  VAR Antwort : CHAR;
  BEGIN
    WriteLn;
    WriteString("Soll das Programm fortgesetzt werden (J/N) ? ");
    REPEAT
      Read(Antwort);
      Antwort:=CAP(Antwort);  (* In Großbuchstaben umwandeln *)
      WriteLn;
      IF (Antwort<>"J") AND (Antwort<>"N")
      THEN WriteString("Bitte nur 'J' oder 'N' ? ")
      END  (* IF *)
    UNTIL (Antwort="J") OR (Antwort="N");
    IF Antwort="J"
    THEN WriteString("Das Programm wird fortgesetzt."); WriteLn
    ELSE WriteString("Das Programm wird abgebrochen."); WriteLn
    END (* IF *)
  END JaNein1.
```

Probelauf:

```
Soll das Programm fortgesetzt werden (J/N) ? x
Bitte nur 'J' oder 'N' ? m
Bitte nur 'J' oder 'N' ? n
Das Programm wird abgebrochen.
```

Eine andere Möglichkeit der Problemlösung besteht darin, die Zeichen unsichtbar einzulesen, bis eine zulässige Taste gedrückt wurde. Eine Prozedur, die das bewerkstelligt, ist unter dem Namen «BusyRead» im Bibliotheksmodul «Terminal» zu finden. Diese Prozedur liefert das Zeichen mit dem ASCII-Wert 0, wenn keine Taste gedrückt wurde, ansonsten das jeweilige Zeichen.

Verwendet man «BusyRead», so muß man das Programm so lange warten lassen, bis eine Taste gedrückt wurde. Auch das geschieht mit einer (verschachtelten) REPEAT-Anweisung:

```
REPEAT BusyRead(Antwort) UNTIL Antwort>0C
```

Hier nun eine zweite Version der Ja/Nein-Abfrage. Diesmal werden unzulässige Eingaben mit einem Piepton (Write(7C)) quittiert. Erst

bei «J» oder «N» erfolgt ein Echo (Ausgabe des getippten Buchstabens) auf dem Bildschirm.

```
MODULE JaNein2;
  FROM InOut IMPORT Write, WriteString, WriteLn;
  FROM Terminal IMPORT BusyRead;
  VAR Antwort : CHAR;
  BEGIN
    WriteLn;
    WriteString("Soll das Programm fortgesetzt werden (J/N) ? ");
    REPEAT
      REPEAT BusyRead(Antwort) UNTIL Antwort>0C;
      Antwort:=CAP(Antwort);  (* In Großbuchstaben umwandeln *)
      IF (Antwort<>"J") AND (Antwort<>"N")
      THEN Write(7C)          (* Läßt den Rechner piepsen *)
      END  (* IF *)
    UNTIL (Antwort="J") OR (Antwort="N");
    Write(Antwort); WriteLn;
    IF Antwort="J"
    THEN WriteString("Das Programm wird fortgesetzt."); WriteLn
    ELSE WriteString("Das Programm wird abgebrochen."); WriteLn
    END (* IF *)
  END JaNein2.
```

Verzichtet man auf das Piepen bei unzulässigen Eingaben, kann das Programm noch einfacher gestaltet werden:

```
MODULE JaNein3;
  FROM InOut IMPORT Write, WriteString, WriteLn;
  FROM Terminal IMPORT BusyRead;
  VAR Antwort : CHAR;
  BEGIN
    WriteLn;
    WriteString("Soll das Programm fortgesetzt werden (J/N) ? ");
    REPEAT
      BusyRead(Antwort)
      Antwort:=CAP(Antwort);  (* In Großbuchstaben umwandeln *)
    UNTIL (Antwort="J") OR (Antwort="N");
    Write(Antwort); WriteLn;
    IF Antwort="J"
    THEN WriteString("Das Programm wird fortgesetzt."); WriteLn
    ELSE WriteString("Das Programm wird abgebrochen."); WriteLn
    END (* IF *)
  END JaNein3.
```

Unser zweites Beispiel stammt aus der Mathematik. Wir wollen die Quadratwurzel einer eingegebenen (REAL-)Zahl bestimmen. Dazu wählen wir die einfache (und uneffektive) Methode der Intervallhalbierung. Das Verfahren funktioniert folgendermaßen:

Die Quadratwurzel einer Zahl liegt auf jeden Fall zwischen 0.0 und dieser Zahl selbst. Also setzen wir die Untergrenze des Intervalls auf 0.0 und die Obergrenze auf die eingegebene Zahl. Dann wird die Mitte des Intervalls ((Obergrenze + Untergrenze)/2) bestimmt.

Ist das Quadrat (Mitte * Mitte) größer als die eingegebene Zahl, so liegt die gesuchte Wurzel in der unteren Intervallhälfte, und wir setzen einfach die Obergrenze auf die Mitte. Andernfalls befindet sich die Wurzel in der oberen Hälfte, und wir legen deshalb die Untergrenze auf die Mitte.

Dieses Verfahren der Intervall-Halbierung wiederholen wir so lange, bis sich das Quadrat der eingegebenen Zahl nur noch um einen sehr kleinen Betrag unterscheidet (ABS(Mitte * Mitte – Eingabe)< KleinerBetrag). Die Mitte ist dann die gesuchte Wurzel (in Annäherung).

```
MODULE Wurzel1;
  FROM InOut IMPORT WriteString, WriteLn;
  FROM RealInOut IMPORT ReadReal, WriteReal;
  VAR Eingabe, Obergrenze, Untergrenze, Mitte : REAL;
  BEGIN
    WriteLn;
    WriteString("Bestimmung der Quadratwurzel einer Zahl"); WriteLn;
    WriteString("---------------------------------------"); WriteLn;
    WriteLn;
    WriteString("Ihre Zahl: "); ReadReal(Eingabe); WriteLn;
    Obergrenze:=Eingabe; Untergrenze:=0.0;
    REPEAT
      Mitte:=(Obergrenze+Untergrenze)/2.0;
      IF Mitte * Mitte > Eingabe
      THEN Obergrenze:=Mitte  (* Wurzel liegt zwischen Untergrenze
                                 und Mitte *)
      ELSE Untergrenze:=Mitte (* Wurzel liegt zwischen Mitte
                                 und Obergrenze *)
      END (* IF *)
    UNTIL ABS(Mitte*Mitte-Eingabe)<0.000001;
    WriteString("Die Wurzel ist ");
    WriteReal(Mitte,20);
    WriteLn
  END Wurzel1.
```

Probelauf:

```
Bestimmung der Quadratwurzel einer Zahl
---------------------------------------

Ihre Zahl: 2
Die Wurzel ist 1.4142284737764E0
```

Dieses Programm funktioniert zwar, hat aber noch einige Schönheitsfehler:

- Wenn keine gültige REAL-Zahl eingegeben wurde, ist die Variable «Eingabe» nicht definiert. Das Ergebnis ist unsinnig.

- ☐ Falls der Anwender eine negative Zahl eingibt, funktioniert das Verfahren nicht – es kommt zu einer Endlosschleife (d. h., der boolesche Ausdruck wird niemals «TRUE»).
- ☐ Es ist sehr ungünstig, wenn im Anweisungsteil eines Programms Zahlenkonstante auftreten. Solche Programme sind immer schwer zu warten. Bei einer Änderung (um z. B. auf eine höhere Genauigkeit umzustellen) muß der gesamte Programmtext, der mitunter sehr groß sein kann, nach dem Vorkommen solcher Zahlenkonstanten untersucht werden.

Doch sind diese Fehler leicht auszumerzen. Das Modul «RealInOut» exportiert die boolesche Variable «Done». Diese gibt im Anschluß an eine Zahleneingabe Aufschluß darüber, ob die Operation erfolgreich war (TRUE) oder nicht (FALSE). Wir können also die Eingabe so lange wiederholen, bis uns die Variable «Done» die Gültigkeit der Eingabe anzeigt.

Negative Zahlen können wir selbst sehr leicht abfangen. Wir müssen uns nur überlegen, wie unser Programm reagieren soll. Eigentlich sind Quadratwurzeln aus negativen Zahlen im Bereich der reellen Zahlen nicht darstellbar. Wenn wir aber in einem solchen Fall die Wurzel aus dem Absolutbetrag der Zahl ziehen und das Ergebnis mit «i» (Wurzel aus -1) multiplizieren, erhalten wir als Ergebnis eine komplexe Zahl. Das Programm muß sich also nur merken, ob die Zahl negativ oder positiv war. Hier bietet sich eine Variable «negativ» vom Typ «BOOLEAN» an, die nach einer gültigen Eingabe entsprechend gesetzt wird:

```
IF Eingabe<0.0 THEN negativ:=TRUE ELSE negativ:=FALSE
END
```

Diese Anweisung kann jedoch mit einer Zuweisung wesentlich effektiver ausgedrückt werden:

```
negativ:=Eingabe<0.0
```

Um den Programmtext von Konstanten freizuhalten, besteht in Modula-2 die Möglichkeit, diese – mit einem Namen versehen – an den Anfang des Programms (genauer: in den Deklarationsteil) zu stellen. Man spricht hier von einer Konstanten-Vereinbarung. Die Syntax ist:

```
Konstanten-Vereinbarung::="CONST" {Name "=" Konstanter-
Ausdruck ";"}.
```

Dabei darf der Ausdruck selbst nur Konstante (auch bereits vereinbarte) enthalten, nicht jedoch Variable oder Funktionen. Normalerweise ist der Typ der Konstanten nach der Auswertung des Ausdrucks bestimmt.

- ☐ REAL-Konstante werden mit Dezimalpunkt und (bei Bedarf) mit Exponent geschrieben.
- ☐ Ganze Zahlen von MAX(INTEGER)+1 bis MAX(CARDINAL) werdendem Typ CARDINAL zugeordnet.
- ☐ Negative ganze Zahlen werden als INTEGER-Konstante interpretiert.
- ☐ Positive Konstante von 0 bis MAX(INTEGER) können sowohl in INTEGER- als auch in CARDINAL-Ausdrücken verwendet werden.

Beispiele:

```
CONST   pi = 3.1415;
        Schranke = 0.000001;
        MaxZahl = 1000;
        Groesse = MaxZahl * pi;
        Grenze = 1.E-10;
        Ja = "J";
        Wahr = TRUE;
```

Der Einsatz der Konstanten-Vereinbarung bringt ausschließlich Vorteile:

- ☐ Klarheit durch «sprechende» Namen,
- ☐ Wartbarkeit durch einfache Änderung an einer Stelle im Programm.

```
MODULE Wurzel2;
  FROM InOut IMPORT WriteString, WriteLn;
  FROM RealInOut IMPORT ReadReal, WriteReal, Done;
  CONST Grenze = 0.000001;
  VAR Eingabe, Obergrenze, Untergrenze, Mitte : REAL;
      negativ : BOOLEAN;
  BEGIN
    WriteLn;
    WriteString("Bestimmung der Quadratwurzel einer Zahl"); WriteLn;
    WriteString("---------------------------------------"); WriteLn;
    WriteLn;
    REPEAT
      WriteString("Ihre Zahl: "); ReadReal(Eingabe); WriteLn
    UNTIL Done;
    negativ:=Eingabe<0.0;       (* Vorzeichen merken *)
```

```
    Eingabe:=ABS(Eingabe);      (* Mit Absolutwert weiterrechnen *)
    Obergrenze:=Eingabe; Untergrenze:=0.0;
    REPEAT
      Mitte:=(Obergrenze+Untergrenze)/2.0;
      IF Mitte * Mitte > Eingabe
      THEN Obergrenze:=Mitte  (* Wurzel liegt zwischen Untergrenze
                                 und Mitte *)
      ELSE Untergrenze:=Mitte (* Wurzel liegt zwischen Mitte
                                 und Obergrenze *)
      END (* IF *)
    UNTIL ABS(Mitte*Mitte-Eingabe)<Grenze;
    WriteString("Die Wurzel ist ");
    WriteReal(Mitte,20);
    IF negativ THEN WriteString(" * i") END;
    WriteLn
  END Wurzel2.
```

Testlauf:

```
Bestimmung der Quadratwurzel einer Zahl
---------------------------------------------------------
Ihre Zahl:    Computer sind doof!
Ihre Zahl:    Mag nicht
Ihre Zahl:    -13
Die Wurzel ist 3.605551258912E0 * i
```

Aufgaben:

1. Auch mit CARDINAL-Zahlen können Dezimalbrüche dargestellt werden. Man muß dabei nur wie bei der schriftlichen Teilung zweier Zahlen vorgehen. Das Ergebnis von «A geteilt durch B»kann so ermittelt werden:

   Vorkommazahl:= A DIV B;
   Rest:=A MOD B;
   1. Nachkommastelle := (10 * Rest) DIV B;
   Rest:=(10 * Rest) MOD B;
   2. Nachkommastelle := (10 * Rest) DIV B;
   Rest:=(10 * Rest) MOD B
   usw.

   Schreiben Sie ein Programm, das eine Division auf 10 Stellen genau durchführt.
2. Schreiben Sie ein Programm, das eine Folge von CARDINAL-Zahlen einliest (durch 0 abgeschlossen) und den Durchschnittswert (auf zwei Stellen genau) dieser Folge ausgibt.

4.3 Die WHILE-Anweisung

Wenn die Schleife abweisend sein soll, wenn also die Bedingung (boolescher Ausdruck) bereits am Anfang der Schleife geprüft werden soll, so kann das mit der REPEAT-Anweisung in Verbindung mit einer IF-Anweisung realisiert werden:

```
IF NOT BoolescherAusdruck
THEN REPEAT
       Anweisungssequenz
     UNTIL BoolescherAusdruck
END
```

Da dieser Fall in der Programmierpraxis sehr häufig vorkommt, gibt es für diese Konstruktion eine eigene Anweisung:

WHILE-Anweisung::="WHILE" BoolescherAusdruck "DO" Anweisungssequenz "END".

Hier wird die Anweisungssequenz so lange wiederholt, wie der boolesche Ausdruck den Wert «TRUE» liefert. Da die erste Prüfung bereits vor dem ersten Durchlauf stattfindet, kann die Anweisungssequenz u.U. überhaupt nicht ausgeführt werden.

Beispiel: Ein Programm soll die Fakultät einer eingegebenen CARDINAL-Zahl berechnen. Die Fakultät ist das Produkt der natürlichen Zahlen von 1 bis zur eingegeben Zahl, also 1 * 2 * 3 * ... *n.

```
MODULE Fakultaet;
  FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn, Done;
  BAR Eingabe, Fakultaet : CARDINAL;
  BEGIN
   WriteLn;
   WriteString("Fakultäts-Berechnung"); WriteLn;
   WriteString("--------------------"); WriteLn;
   WriteLn;
   REPEAT
     WriteString("Bitte geben Sie eine natürliche Zahl ein: ");
     ReadCard(Eingabe); WriteLn
   UNTIL Done;
   Fakultaet:=1;
   WHILE Eingabe>0 DO
     Fakultaet:=Fakultaet*Eingabe;
     Eingabe:=Eingabe-1
   END; (* WHILE *)
   WriteString("Die Fakultät ist: ");
   WriteCard(Fakultaet,10);
   WriteLn
  END Fakultaet.
```

Probelauf:

```
Fakultäts-Berechnung
--------------------

Bitte geben Sie eine natürliche Zahl ein: 6
Die Fakultät ist: 720
```

Analysieren Sie bitte das Programm ganz genau. Die Variable «Done» aus dem Modul «InOut» wird erst dann «TRUE», wenn eine gültige CARDINAL-Zahl eingegeben wurde. Dann wird die Variable «Fakultaet» auf den Wert «1» gesetzt. Wurde «0» eingegeben, so wird die Anweisungssequenz der WHILE-Anweisung nicht durchlaufen. Das Ergebnis ist dann «1», ein mathematisch korrektes Resultat.

Spielen wir die WHILE-Anweisung einmal mit der Eingabe «5» durch:

Bedingung	Fakultät	Eingabe
	1	5 (Beim Eintritt)
5>0 (TRUE)	1*5 (=5)	5-1 (=4)
4>0 (TRUE)	5*4 (=20)	4-1 (=3)
3>0 (TRUE)	20*3 (=60)	3-1 (=2)
2>0 (TRUE)	60*2 (=120)	2-1 (=1)
1>0 (TRUE)	120*1 (=120)	1-1 (=0)
0>0 (FALSE)		(Anweisung beendet)

In einem zweiten Beispiel wollen wir von der sog. Eingabe-Umleitung (engl. input-redirection) Gebrauch machen. Das Modul «InOut» liefert die dazu nötigen Prozeduren «OpenInput» und «CloseInput». «OpenInput» muß ein Zeichenketten-Parameter mitgegeben werden. Es handelt sich dabei um die Extension des Dateinamens, wenn diese nicht explizit bei der Abfrage angegeben wird.

Der Aufruf von «OpenInput(«MOD»)» bsw. bewirkt, daß das Programm auf dem Bildschirm eine Meldung wie

```
INPUT FROM      (Eingabe von) oder
in>             (ein>)
```

ausgibt. Dann wird die Eingabe eines Dateinamens erwartet. Wird dabei keine Extension angegeben, wird automatisch «.MOD» ange-

hängt. Im Anschluß daran wird versucht, diese Datei zum Lesen zu öffnen. Ist dieser Versuch erfolgreich, wird die Variable «Done» auf «TRUE» gesetzt, andernfalls auf «FALSE». Jede weitere Eingabe wird in der Folge aus dieser Datei geholt, wobei «Done» nach jeder Operation über den Erfolg Aufschluß gibt. Am Dateiende wird «Done» auf jeden Fall «FALSE». Nach «CloseInput» werden alle weiteren Eingaben wieder von der Tastatur geholt.

Wir haben es hier zum ersten Mal mit der Bearbeitung einer externen Datei zu tun. Die Umleitung ist auf alle Kanäle möglich, die – wie die Tastatur – Daten vom Typ «CHAR» liefern. Man bezeichnet solche Dateien auch als «Textdateien», den Zeichenstrom, den sie liefern, als «Text».

Das Beispielprogramm erfüllt eine einfache Aufgabe: Es zählt die Buchstaben, aus denen der gelieferte Text besteht (daher der Modulname «Count = Zählen»). Da es aber das Grundgerüst für viele weitere Beispiele enthält, sollten Sie es vollkommen durchschauen.

```
MODULE Count;
  FROM InOut IMPORT OpenInput, CloseInput, Read, WriteLn,
                    WriteCard, WriteString;
  VAR Zeichen : CHAR;
      BuchstabenZahl : CARDINAL;
  BEGIN
   WriteLn;
   WriteString("Anzahl der Buchstaben in einem Text"); WriteLn;
   WriteString("-----------------------------------"); WriteLn;
   WriteLn;
   OpenInput("MOD"); WriteLn;
   IF NOT Done
   THEN WriteString("Datei existiert nicht!"; WriteLn; HALT
   END; (* IF *)
   BuchstabenAnzahl:=0;
   Read(Zeichen);
   WHILE Done DO
     Zeichen:=CAP(Zeichen);
     IF (Zeichen>="A") AND (Zeichen<="Z")
     THEN BuchstabenAnzahl:=BuchstabenAnzahl+1
     END; (* IF *)
     Read(Zeichen)
   END; (* WHILE *)
   CloseInput;
   WriteString("Analyse beendet."); WriteLn;
   WriteString("Es wurden ");
   WriteCard(BuchstabenAnzahl,1);
   WriteString(" Buchstaben gefunden.");
   WriteLn
  END Count.
```

Probelauf:

```
Anzahl der Buchstaben in einem Text
---------------------------------------------------

INPUT FROM COUNT
Analyse beendet.
Es wurden 547 Buchstaben gefunden.
```

Aus diesem Programm läßt sich mit wenigen Handgriffen ein Kopierprogramm für Textdateien erstellen. Dazu leiten wir – entsprechend der Eingabe-Umleitung – auch die Ausgabe um. Die benötigten Prozeduren «OpenOutput» und «CloseOutput» finden wir wiederum in «InOut». Sie entsprechen völlig den schon bekannten Prozeduren «OpenInput» und «CloseInput».

```
MODULE Kopieren;
  FROM InOut IMPORT Read, Done, OpenInput, CloseInput, OpenOutput,
                    CloseOutput, Write, WriteLn, WriteString;
  VAR Zeichen : CHAR;
  BEGIN
   WriteLn;
   WriteString("Kopieren von Text-Dateien"); WriteLn;
   WriteString("-------------------------"); WriteLn;
   WriteLn;
   OpenInput("MOD"); WriteLn;
   IF NOT Done
   THEN WriteString("Datei existiert nicht!"; WriteLn; HALT
   END; (* IF *)
   OpenOutput("MOD");
   Read(Zeichen);
   WHILE Done DO
     Write(Zeichen);
     Read(Zeichen)
   END; (* WHILE *)
   CloseInput; CloseOutput;
   WriteString("Kopie erstellt."); WriteLn;
  END Kopieren.
```

Probelauf:

```
Kopieren von Text-Dateien
-------------------------------------

INPUT FROM KOPIEREN
OUTPUT TO COPY
Kopie erstellt.
```

Aufgaben:

1. Schreiben Sie das Programm unter ausschließlicher Verwendung von REPEAT-Anweisungen.
2. Ändern Sie das Programm «Count» so, daß Groß- und Kleinbuchstaben getrennt gezählt werden.
3. Ändern Sie das Programm «Kopieren» so, daß alle Klein- in Großbuchstaben umgewandelt werden.

4.4 Die LOOP-Anweisung

Schleifen, die immer wieder durchlaufen und niemals abgebrochen werden, nennt man «Endlosschleifen». Normalerweise sind Endlosschleifen grobe Programmierfehler, denen nur schwer auf die Schliche zu kommen ist, da das Programm an dieser Stelle richtiggehend hängenbleibt. Allerdings gibt es einige wenige Fälle, in denen Endlosschleifen angebracht sind. Ein Beispiel hierfür ist das Betriebssystems eines Rechners:

Wiederhole (immer und immer wieder)
Melde Bereitschaft.
Lies Befehl von der Tastatur.
Führe den Befehl aus.

Obwohl Endlosschleifen relativ einfach mit einer WHILE- (WHILE TRUE DO ...) oder einer REPEAT-Anweisung (REPEAT ... UNTIL FALSE) realisiert werden können, bietet Modula-2 in der LOOP-Anweisung eine eigene Endlosschleife. Der Grund hierfür liegt einerseits darin, daß durch den Wegfall des booleschen Ausdrucks die Anweisung schneller abgearbeitet werden kann. Andererseits ist es gerade bei prinzipiell unendlich durchlaufenen Schleifen wichtig, diese Eigenschaft transparent zu machen. Die Syntax der LOOP-Anweisung lautet:

LOOP-Anweisung::="LOOP" Anweisungssequenz "END".

Die von den beiden Schlüsselwörtern «LOOP» und «END» eingeschlossene Anweisungssequenz wird bedingungslos wiederholt. Soll die Schleife dennoch verlassen werden, so wird für diesen Zweck die EXIT-Anweisung bereitgestellt, die nur aus dem Schlüsselwort «EXIT» besteht.

EXIT-Anweisung::="EXIT".

Die EXIT-Anweisung bewirkt, daß das Programm unmittelbar nach der LOOP-Anweisung fortgesetzt wird. Innerhalb einer LOOP-Anweisung können beliebig viele EXIT-Anweisungen vorkommen (außerhalb führt eine EXIT-Anweisung zu einer Fehlermeldung). Diese Möglichkeit erweitert den Einsatzbereich der LOOP-Anweisung auf solche Fälle, bei denen die Schleife infolge selten auftretender Ereignisse abgebrochen werden muß, deren stetige Abfrage in den booleschen Ausdrücken der WHILE- und REPEAT-Anweisung zu einem übermäßigen Rechenaufwand führen würde:

```
LOOP
  IF NochDatenDa
  THEN LiesDaten
  ELSIF FehlerAufgetreten
  THEN EXIT
  ELSE
    OeffneNeueDatei
    IF KeineMehrDa THEN EXIT END
  END
END (* LOOP *)
```

Würde man diese Anweisung mit einer REPEAT-Anweisung verwirklichen, so würde die direkte Umsetzung dazu führen, daß nach jedem Schleifendurchlauf alle drei Bedingungen geprüft werden müssen:

```
REPEAT
  IF NochDatenDa
  THEN LiesDaten
  ELSIF NOT FehlerAufgetreten
  THEN OeffneNeueDatei
  END (* IF *)
UNTIL (KeineDatenMehrDa AND KeineDateiMehrda) OR FehlerAufgetreten
```

Um das zu vermeiden, benutzt man meist eine boolesche Variable, die über das Schleifenende Auskunft gibt:

```
VAR Fertig : BOOLEAN;
...
Fertig:=FALSE;
REPEAT
  IF NochDatenDa
  THEN LiesDaten
  ELSIF FehlerAufgetreten
  THEN Fertig:=TRUE
  ELSE
    OeffneNeueDatei
```

```
    IF KeineMehrDa THEN Fertig:=TRUE
  END (* IF *)
UNTIL Fertig
```

Es ist Geschmackssache, ob man dieser Konstruktion den Vorzug gegenüber der Variante mit der LOOP-Anweisung gibt. Schlechte Programmierpraxis ist es jedoch, wenn man mit der LOOP-Anweisung die WHILE- und/oder die REPEAT-Anweisung ersetzt:

WHILE BoolescherAusdruck DO Anweisungssequenz END <–>

LOOP IF NOT BoolescherAusdruck THEN EXIT END;
Anweisungssequenz END

REPEAT Anweisungssequenz UNTIL BoolescherAusdruck <–>

LOOP Anweisungssequenz; IF BoolescherAusdruck
THEN EXIT END

Obwohl sich die Anweisungen jeweils genau gleich verhalten, geht bei den LOOP-Varianten die Klarheit und damit Information für den Menschen verloren. Die LOOP-Anweisung soll also wirklich nur dann eingesetzt werden, wenn

- ☐ eine echte Endlosschleife benötigt wird oder
- ☐ der Einsatz zu einer Effizienzsteigerung führt, die auf anderem Wege nicht erreichbar ist.

Hinweis: Bei Coroutinen und parallelen Prozessen werden häufig Endlosschleifen benötigt. Hier liegt auch das Haupteinsatzgebiet für die LOOP-Anweisung.

Aufgaben:

1. Schreiben Sie das Fakultätprogramm unter Verwendung der LOOP-Anweisung.
2. Schreiben Sie das Text-Kopierprogramm mit der LOOP-Anweisung.

4.5 Die FOR-Anweisung

Ein häufiger Sonderfall liegt bei solchen Schleifen vor, bei denen die Anzahl der Wiederholungen von vornherein bekannt ist. Hier kommt die FOR-Anweisung zum Einsatz:

FOR-Anweisung::="FOR" Laufvariable ":=" Startwert "TO"
Endwert
["BY" Schrittweite] "DO" Anweisungssequenz
"END".

Startwert und Endwert sind Ausdrücke vom selben aufzählbaren Typ wie die Laufvariable. Die optionale Schrittweite ist eine Konstante vom Typ INTEGER oder CARDINAL. Wenn die Schrittweite fehlt, wird 1 angenommen. Bei der Ausführung der FOR-Anweisung geschieht nun folgendes:

1. Die Laufvariable (muß wie alle Variablen deklariert sein) wird auf den Startwert gesetzt.
2. Der Wert der Laufvariablen wird mit dem Endwert verglichen. Ist dieser überschritten (bzw. unterschritten bei negativer Schrittweite), so ist die FOR-Anweisung beendet.
3. Die Anweisungssequenz wird ausgeführt. Innerhalb der Anweisungssequenz kann auf die Laufvariable zugegriffen werden. Sie darf jedoch keinesfalls verändert werden!
4. Zur Laufvariablen wird um die Schrittweite erhöht (bzw. erniedrigt bei negativer Schrittweite) und das Verfahren bei 2. fortgesetzt.

Da der Vergleich immer vor der Anweisungssequenz stattfindet, ist die FOR-Schleife abweisend. Das folgende Beispiel zählt auf dem Bildschirm bis 100.

```
MODULE HeraufZaehlen;
  FROM InOut IMPORT WriteCard, WriteLn;
  VAR Zaehler : CARDINAL;
  BEGIN
    FOR i:=1 TO 100 DO
      WriteCard(Zaehler,10);
      WriteLn
    END (* FOR *)
  END HeraufZaehlen.
```

Probelauf:

```
1
2
3
4
...
```

Im nächsten Beispiel wird in Zweierschritten von 100 bis 0 gezählt:

```
MODULE HerunterZaehlen;
  FROM InOut IMPORT WriteCard, WriteLn;
  VAR Zaehler : CARDINAL;
  BEGIN
    FOR i:=100 TO 0 BY -2 DO
      WriteCard(Zaehler,10);
      WriteLn
    END (* FOR *)
  END HerunterZaehlen.
```

Probelauf:

```
100
 98
 96
...
```

Mit der FOR-Anweisung läßt sich die Fakultät ganz einfach berechnen:

```
MODULE FakultaetMitFOR;
  FROM InOut IMPORT ReadCard, WriteCard, WriteLn, WriteString;
  VAR Eingabe, Fakultaet, Zaehler : CARDINAL;
  BEGIN
    WriteString("Berechnung der Fakultät mit der FOR-Anweisung"); WriteLn;
    WriteString("-----------------------------------------"); WriteLn;
    WriteLn;
    WriteString("Bitte eine natürliche Zahl: ");
    ReadCard(Eingabe); WriteLn;
    Fakultaet:=1;  (* Anfangswert *)
    FOR Zaehler:=1 TO Eingabe DO
      Fakultaet:=Fakultaet*Zaehler
    END; (* FOR *)
    WriteCard(Eingabe,1);
    WriteString("! = ");
    WriteCard(Fakultaet,1);
    WriteLn
  END FakultaetMitFOR.
```

Probelauf:

```
Berechnung der Fakultät mit der FOR-Anweisung
---------------------------------------------------------------------

Bitte eine natürliche Zahl: 7
7! = 5040
```

Hinweis: n! ist die mathematische Schreibweise für die Fakultät von n.

Es bedarf hoffentlich keiner besonderen Betonung mehr, daß FOR-Anweisungen (wie alle strukturierten Anweisungen) beliebig verschachtelt werden können. Das folgende Programm gibt eine ASCII-Tabelle aller druckbaren Zeichen aus. Dabei greifen wir auf die Konvention zurück, daß Laufvariable in Programmen mit «i» und «j» bezeichnet werden. Da die CHARACTER-Konstanten in Modula-2 im Oktalsystem angegeben werden müssen, geben wir die Spalten- und Zeileninformation auch in diesem Format aus. Dazu importieren wir die Prozedur «WriteOct» aus dem Standardmodul «InOut».

```
MODULE ASCIITabelle;
  FROM InOut IMPORT Write, WriteString, WriteLn, WriteOct;
  VAR i,j : CARDINAL;
  BEGIN;
    WriteLn;
    WriteString("Die ASCII-Tabelle"); WriteLn;
    WriteString("-----------------"); WriteLn;
    WriteLn;
    WriteString("  ");
    FOR i:=0 TO 7 DO WriteOct(i,3) END; (* Spaltenüberschrift *)
    WriteLn;
    FOR i:=32 TO 127 BY 8 DO
      WriteOct(i,3);  (* Zeilen-Nummer *)
      FOR j:=0 TO 7 DO
        Write(" ");
        Write(CHR(i+j));
        Write(" ")
      END; (* FOR j *)
      WriteLn
    END (* FOR i *)
  END ASCIITabelle.
```

Probelauf:

Die ASCII-Tabelle

	0	1	2	3	4	5	6	7
40		!	"	#	$	%	&	'
50	(	)	*	+	,	-	.	/
60	0	1	2	3	4	5	6	7
70	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	^	
140	'	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	|	}	~	

Für die Laufvariable (sowie den Start- und Endwert) gibt es nur die Einschränkung, daß sie demselben aufzählbaren Typ angehören müssen. Demnach kann die Laufvariable beispielsweise auch vom Typ CHAR oder BOOLEAN sein.

Das folgende Beispiel schreibt erst alle Großbuchstaben aufsteigend und dann alle Kleinbuchstaben absteigend auf den Bildschirm:

```
MODULE Buchstaben;
  FROM InOut IMPORT Write, WriteLn;
  VAR Zeichen : CHAR;
  BEGIN
    WriteString("Die Buchstaben: "); WriteLn; WriteLn;
    FOR Zeichen:="A" TO "Z" DO Write(Zeichen);
    WriteLn;
    FOR Zeichen:="z" TO "a" BY -1 DO Write(Zeichen);
    WriteLn
  END Buchstaben.
```

Probelauf:

Die Buchstaben:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
zyxwvutsrqponmlkjihgfedcba

Gleich dreifach verschachtelt ist die FOR-Anweisung im nächsten Beispiel. Es handelt sich dabei um ein Programm, das eine einfache Knobelaufgabe löst:

Ein Weg gabelt sich in drei Richtungen (A, B und C). Eine Familie, die mit den örtlichen Gegebenheiten nicht vertraut ist, weiß nun nicht, welcher Weg der richtige zur Fortsetzung ihrer Reise ist. Glücklicherweise steht an der Weggabel ein Einheimischer. Doch leider beantwortet er die Frage nach dem richtigen Weg genauso verklausuliert wie einst das Orakel von Delphi:

1. Wenn der Weg B falsch ist, dann sind sowohl A als auch C richtig.
2. Wenn entweder A oder B richtig ist, führt C in die Irre.
3. Wenn C oder A richtig ist, dann ist B falsch.

Die Lösung:

Zur Ermittlung des richtigen Weges verwenden wir die Brute-Force-Methode (Methode der rohen Gewalt). Dabei werden alle Möglichkeiten durchgeprüft. Wenn bei einer Konstellation alle drei Sätze wahr werden, haben wir eine Lösung gefunden.

Eine Richtung kann hier nur die Werte «richtig» oder «falsch» annehmen, damit ist der Datentyp BOOLEAN angebracht. Erinnern wir uns an das Kapitel über den Datentyp BOOLEAN. Die Implikation (wenn ... dann) wird am einfachsten durch <= ausgedrückt. Den ersten Satz kann man so bilden:

Satz1 := (NOT B)<= (A AND C)

Genauso können die Sätze 2 und 3 umgesetzt werden:

Satz2 := (A<>B) <= (NOT C)
Satz3 := (C OR A) <= (NOT B)

Wirklich erschöpfend ist unsere Prüfung, wenn wir alle TRUE-FALSE-Kombinationen berechnen.

```
MODULE Knobelei;
  FROM BooleanInOut IMPORT WriteBoolean;
  FROM InOut IMPORT WriteString, WriteLn;
  VAR A,B,C,  (* Die Richtungen *)
      Satz1, Satz2, Satz3 (* die Sätze *) : BOOLEAN;
```

```
BEGIN
  WriteString("Lösung der Knobelaufgabe:"); WriteLn;
  WriteLn;
  FOR A:=FALSE TO TRUE DO
    FOR B:=FALSE TO TRUE DO
      FOR C:=FALSE TO TRUE DO
        Satz1:=(NOT B)<=(A AND C);
        Satz2:=(A<>B)<=(NOT C);
        Satz3:=(C OR A)<=(NOT B);
        IF Satz1 AND Satz2 AND Satz3
        THEN
          WriteString("A -> "); WriteBoolean(A); WriteLn;
          WriteString("B -> "); WriteBoolean(B); WriteLn;
          WriteString("C -> "); WriteBoolean(C); WriteLn;
          WriteLn
        END (* IF *)
      END (* FOR C *)
    END (* FOR B *)
  END (* FOR A *)
END Knobelei.
```

Probelauf:

Lösung der Knobelaufgabe:

A -> FALSE
B -> TRUE
C -> FALSE

Tip zur Textgestaltung mit Wiederholungsanweisungen: Falls die gesamte Schleife nicht in eine Zeile paßt, ist es vorteilhaft, die Anweisungssequenz gegenüber dem Schleifenkopf (WHILE, REPEAT, LOOP, FOR) einzurücken und das Schleifenende (END, UNTIL) wieder auszurücken.

Aufgaben:

1. Schreiben Sie ein Programm, das die Summe aller natürlichen Zahlen bis zu einer eingegebenen CARDINAL-Zahl berechnet und ausgibt.
2. Mit der Standardprozedur «FLOAT(CardinalZahl)» kann eine CARDINAL- in eine REAL-Zahl umgewandelt werden. Schreiben Sie ein Programm, das eine positive REAL-Zahl einliest und – mit einer FOR-Anweisung – die kleinste CARDINAL-Zahl sucht, die gleich oder größer der eingegebenen REAL-Zahl ist.
3. Lassen Sie den Quotient zweier eingegebener CARDINAL-Zahlen in Form einer Dezimalzahl auf n Stellen genau ausgeben (wie Aufgabe 1 aus Abschnitt 4.2). Benutzen Sie dazu die FOR-Anweisung.

4.6 Die CASE-Anweisung

Angenommen, Sie wollen ein Programm schreiben, mit dem ein Text so umkopiert wird, daß die deutschen Sonderzeichen nach folgenden Regeln entfernt werden:

Ä –> Ae Ö –> Oe Ü –> Ue ä –> ae ö –> oe ü –> ue ß –> ss

Mit den bisher bekannten Werkzeugen könnte das Problem so gelöst werden:

```
IF Zeichen='Ä' THEN WriteString("Ae")
ELSIF Zeichen='Ö' THEN WriteString("Oe")
ELSIF Zeichen='Ü' THEN WriteString("Ue")
ELSIF Zeichen='ä' THEN WriteString("ae")
ELSIF Zeichen='ö' THEN WriteString("oe")
ELSIF Zeichen='ü' THEN WriteString("ue")
ELSIF Zeichen='ß' THEN WriteString("ss")
ELSE Write(Zeichen)
END
```

Da solche Problemstellungen in der Praxis sehr häufig sind, gibt es in Modula-2 eine ganz wesentliche Vereinfachung für diese Konstruktion.

```
CASE-Anweisung::="CASE" Ausdruck "OF" CASE-Liste
                        {"|" CASE-Liste}
                        "ELSE" Anweisungssequenz "END".
CASE-Liste::=Marke {"," Marke} ":" Anweisungssequenz.
Marke:=KonstanterAusdruck|KonstanterAusdruck ".."
KonstanterAusdruck.
```

Mit der CASE-Anweisung sieht obige Konstruktion so aus:

```
CASE Zeichen OF
  'Ä' : WriteString("Ae")
| 'Ö' : WriteString("Oe")
| 'Ü' : WriteString("Ue")
| 'ä' : WriteString("ae")
| 'ö' : WriteString("oe")
| 'ü' : WriteString("ue")
| 'ß' : WriteString("ss")
ELSE Write(Zeichen)
END
```

Bei der Ausführung der CASE-Anweisung wird zunächst der Ausdruck berechnet. Dann werden die einzelnen CASE-Listen überprüft, ob sich der Wert des Ausdrucks unter den Marken befindet. Fällt ein solcher Vergleich positiv aus, wird die entsprechende Anweisungssequenz abgearbeitet und die CASE-Anweisung beendet. Paßt der Wert des Ausdrucks auf keine Marke, wird die Anweisungssequenz nach «ELSE» ausgeführt.

Der Ausdruck und die konstanten Ausdrücke der einzelnen CASE-Labels müssen vom gleichen aufzählbaren Typ sein. Die Form «Konstante..Konstante» ist eine Abkürzung für alle Werte innerhalb und einschließlich der beiden Konstanten. «1..5» ist demnach eine Abkürzung für «1, 2, 3, 4, 5». Man spricht in diesem Zusammenhang auch vom «Bereich der Zahlen von 1 bis 5».

Die CASE-Anweisung wird nicht nur schneller ausgeführt als eine entsprechende IF-Anweisung, sie ist zudem kürzer und klarer. Dieser Sachverhalt tritt offen zutage, wenn man das obige Problem so erweitert, daß die Umlaute Ä, Ö und Ü nur dann nach der angegebenen Regel umgeformt werden, wenn der folgende Buchstabe kein Großbuchstabe ist. Andernfalls gilt folgende Regel:

Ä –> AE Ö –> OE Ü –> UE (Umlaute in großgeschriebenen Wörtern)

```
MODULE Umlaute;

  FROM InOut IMPORT Write, WriteString, Read, OpenInput, CloseInput,
                    OpenOutput, CloseOutput, Done;

  VAR  Zeichen, Naechstes : CHAR;

BEGIN
  WriteString("Umlaut-Umwandlung"); WriteLn; WriteLn;
  OpenInput(""); IF NOT Done THEN WriteString("Keine Datei!") END;
  OpenOutput("");
  Read(Zeichen);
  WHILE Done DO
    CASE Zeichen OF
     'Ä','Ö','Ü' : Read(Naechstes);
                   IF (Naechstes>='A') AND (Naechstes<='Z')
                   THEN CASE Zeichen OF
                           'Ä' : WriteString("AE")
                         | 'Ö' : WriteString("OE")
                         | 'Ü' : WriteString("UE")
                        END
                   ELSE CASE Zeichen OF
                           'Ä' : WriteString("Ae")
                         | 'Ö' : WriteString("Oe")
                         | 'Ü' : WriteString("Ue")
                        END
                   END; (* IF *)
                   Write(Naechstes)
```

```
        | 'ä' : WriteString("ae")
        | 'ö' : WriteString("oe")
        | 'ü' : WriteString("ue")
        | 'ß' : WriteString("ss")
      ELSE Write(Zeichen)
      END; (* CASE *)
      Read(Zeichen)
    END; (* WHILE *)
    CloseOutput;
    CloseInput
  END Umlaute.
```

Für den Testlauf wurde dieser kurze Text unter dem Namen «TEST» auf Diskette gespeichert:

UMLAUT-VERÄNDERUNG

Während die Umlaute in großgeschriebenen Wörtern (wie in Überschriften) in Großbuchstaben übersetzt werden, wird in anderen Fällen ein «e» angehängt.

Probelauf:

Umlaut-Umwandlung

INPUT FROM TEST
OUTPUT TO CON:

UMLAUT-VERAENDERUNG

Waehrend die Umlaute in grossgeschriebenen Woertern (wie in Ueberschriften) in Grossbuchstaben uebersetzt werden, wird in anderen Faellen ein «e» angehaengt.

Aufgaben:

1. Das Programm «Umlaute» geht davon aus, daß ein Text nicht mit einem der Zeichen Ä, Ö oder Ü endet. Andernfalls würde mit dem Prozeduraufruf «Read(Naechstes)» über das Ende der Datei hinaus gelesen. Ändern Sie das Programm so ab, daß auch dieser Sonderfall korrekt behandelt wird.
2. Schreiben Sie ein Programm, das in einem Text alle Vorkommen von Großbuchstaben, Kleinbuchstaben, Steuer- und Sonderzeichen zählt.
3. Überlegen Sie, in welchen Fällen eine IF-Anweisung nicht durch eine CASE-Anweisung ersetzt werden kann.

5 Prozeduren

Wir haben mit den Anweisungen das elementare Programmierwerkzeug kennengelernt. Aber genausowenig, wie sich ein Maschinenbau-Ingenieur beim Entwurf einer Maschine Gedanken über benötigte Schrauben oder zu verwendende Lötkolben macht, wird ein Programmierer bei der Entwicklung eines komplexen Programms den vorteilhaften Gebrauch dieser oder jener Anweisung ins Kalkül ziehen. Beide werden zunächst komplette Funktionseinheiten als gegeben annehmen, die entweder bereits vorhanden sind oder deren Realisierung später erfolgen wird.

Für den angehenden Programmierer ist es sehr wichtig, sich möglichst früh vom «Denken in Anweisungen» zu lösen. Er vermeidet dadurch von vornherein, sich in Detailaufgaben zu verzetteln und den Überblick über das Gesamtprojekt zu verlieren. Modula-2 unterstützt die Zusammenstellung eines Programms aus Funktionseinheiten wie kaum eine andere Programmiersprache durch das Prozedur- und das Modul-Konzept.

5.1 Die Arbeitsorganisation in einem Unternehmen

Wenn beispielsweise ein Chef an eine untergeordnete Abteilung eine Arbeit delegiert, so interessiert ihn normalerweise ausschließlich das Ergebnis und nicht, wie dieses Ergebnis zustande kommt. Denn schließlich ist es seine Aufgabe, die Arbeit der Abteilungen zu koordinieren und somit seinen Teil zum Erfolg des Unternehmens beizutragen. Ist die delegierte Arbeit sehr komplex, so wird die betroffene Abteilung ihrerseits weitere spezialisierte Arbeitsgruppen bilden. Andere Abteilungen wiederum werden von mehreren Stellen des Unternehmens genutzt. Irgendwann wird bei dieser Organisationsstruktur eine Ebene erreicht, in der sich die anfallenden Arbeiten übersichtlich, klar und eindeutig beschreiben lassen. Und erst dieser Ebene entspricht die der Anweisungen.

Die Rolle des Programmierers beim Programmentwurf wechselt mit dem Fortschreiten seiner Arbeit. Erst ist er an der Stelle der Unternehmensführung, die die Arbeitsabläufe in sehr komplexen und abstrakten Einheiten beschreibt. Ist das Unternehmen erst einmal auf diesem Niveau definiert, nimmt er die Funktion der Leitung der komplexen Abteilungen wahr und organisiert deren innere Arbeitsabläufe usw., bis schließlich die unterste Ebene erreicht wird. Diese Vorgehensweise heißt «Methode der schrittweisen Verfeinerung» oder auch «Top-down-Methode». Sie hat sich als die effizienteste und sicherste Methode der Programmentwicklung erwiesen.

5.2 Prozeduren als Arbeitseinheiten

Der Arbeitseinheit des Unternehmens entspricht die Prozedur in Modula-2. Mit den Standardprozeduren und den Prozeduren, die aus Bibliotheksmoduln importiert wurden, sind uns bereits solche begegnet, die ihre Arbeit jedem zur Verfügung stellen, der sie benötigt. Jetzt werden wir die Möglichkeit kennenlernen, selbst Prozeduren zu formulieren.

In Modula-2 sind Prozeduren Bestandteil der Deklaration. Somit kann der Deklarationsteil wie folgt erweitert werden:

Deklaration::= {Konstantenvereinbarung | Typendefinition |
Variablendeklaration | Prozedurbeschreibung}.

Die Prozedurbeschreibung selbst hat eine ähnliche Gestalt wie ein Programm-Modul. Das ist nach dem bisher Gesagten auch nicht weiter verwunderlich, da eine Prozedur als Arbeitseinheit nichts anderes als ein Teilprogramm ist.

Prozedurbeschreibung::="PROCEDURE" Prozedurname [formale-
Parameterliste] ";"
Block Prozedurname ";"

Die Unterschiede zu einem Programm-Modul sind folgende:

- An die Stelle des Schlüsselwortes «MODULE» tritt «PROCEDURE»
- Es gibt keine IMPORT-Listen, dafür aber eventuell formale Parameter.
- Die Prozedurbeschreibung wird mit «;» abgeschlossen.

Ansonsten stimmen die Strukturen vollkommen überein. Da der Begriff «Block» auch Bestandteil der Prozedurbeschreibung ist und jeder Block wiederum einen eigenen Deklarationsteil enthalten kann, können Prozeduren beliebig verschachtelt werden. Aus diesem Grund bezeichnet man Modula-2 auch als «blockstrukturierte Programmiersprache».

Wir wollen mit einem ganz einfachen Beispiel beginnen. Eine sehr häufig benötigte Aufgabe ist das Löschen des Bildschirms. Wir wollen dafür eine Prozedur schreiben. Normalerweise wird der Bildschirm gelöscht, indem ein bestimmtes Steuerzeichen (meist ^L oder ^Z) mittels «Write» an die Konsole ausgegeben wird.

```
PROCEDURE LoescheBildschirm;
  CONST LoeschZeichen = 14C;  (* ^L *)
  BEGIN
    write(LoeschZeichen)
  END LoescheBildschirm;
```

Selbstverständlich muß «Write» an dieser Stelle bekannt, d. h. importiert worden sein. In der Folge kann auf «LoescheBildschirm» genauso zugegriffen werden wie auf Standardprozeduren oder solche, die aus Bibliotheksmoduln importiert wurden.

5.3 Lokale und globale Bezeichner

Die wichtigste Eigenschaft einer Prozedurbeschreibung ist, daß alle Bezeichner des Deklarationsteils nach außen nicht sichtbar sind. So kann in unserem obigen Beispiel auf die Konstante «LoeschZeichen» vom Anweisungsteil des Programmoduls nicht zugegriffen werden. «LoeschZeichen» ist nur innerhalb des zu «LoescheBildschirm» gehörenden Blocks bekannt.

Die innerhalb eines Blocks festgelegten Bezeichner heißen deshalb «lokal zu diesem Block». Sie sind in diesem und allen untergeordneten (verschachtelten) Blöcken bekannt. Ist ein Bezeichner in einem Block bekannt und nicht im Deklarationsteil dieses Blocks enthalten, so heißt er «global».

«Lokal» und «global» sind also immer relativ zu einem Block (zu einer Umgebung) zu verstehen. So sind beispielsweise die Bezeichner eines Programm-Moduls zu allen Prozeduren als global, zum Modul selbst aber als lokal zu betrachten.

Tritt eine Namenskollision auf, gibt es also zu einem lokalen Bezeichner einen gleichen globalen, so gilt an dieser Stelle nur der

lokale; der globale wird also innerhalb des Blocks verdrängt und ist nicht sichtbar.

Innerhalb eines Blocks ist der Prozedurname selbst als globaler Bezeichner bekannt, der im umgebenden Block festgelegt wurde:

```
MODULE LokalUndGlobal;

  VAR  A, B, C : CARDINAL;

  PROCEDURE P1;
  VAR A : CHAR;
  BEGIN
    (* Hier bekannte Bezeichner:
         A            von P1
         B, C, P1     von LokalUndGlobal *)
  END P1;

PROCEDURE P2
  VAR B, C : REAL;

  PROCEDURE P2a;
    VAR A, B : CARDINAL;
    BEGIN
      (* Hier bekannte Bezeichner:
           P1, P2  von LokalUndGlobal
           A, B    von P2a;
           C       von P2            *)
    END P2a;

  BEGIN  (* Anweisungsteil von P2 *)
    (* Hier bekannte Bezeichner:
         A, P1, P2      von LokalUndGlobal
         B, C, P2a      von P2  *)
  END P2;

BEGIN  (* Anweisungsteil von LokalUndGlobal *)
  (* Hier bekannte Bezeichner:
       A, B, C, P1, P2  von LokalUndGlobal *)
END LokalUndGlobal.
```

Auf die Bedeutung, die das Verständnis von lokalen und globalen Größen für die weitere Programmierarbeit hat, kann gar nicht nachdrücklich genug hingewiesen werden.

Aufgaben:

1. Machen Sie – wie im obigen Beispiel – die gültigen Bezeichner kenntlich.

```
MODULE M;
  VAR X,Y : CARDINAL;
      Z   : REAL;

PROCEDURE P;
  VAR A,B,X,Y : INTEGER;
      Z : CARDINAL;
```

```
PROCEDURE P1;
  VAR A,B : CHAR
  BEGIN
  END P1;

PROCEDURE P2;
  VAR P : CARDINAL;
      X : REAL;

  PROCEDURE P;
    VAR A,B : REAL;
    BEGIN
    END P;

  BEGIN
  END P2;

BEGIN
END M.
```

2. Welche Ausgaben hat das folgende Programm?

```
MODULE LokalTest;
  FROM InOut IMPORT WriteCard, WriteLn;

  VAR A,B, Summe : CARDINAL;

  PROCEDURE P;
    CONST A = 10;
    BEGIN
      WriteCard(B,10); WriteLn;
      B:=2; Summe:=A+B;
      WriteCard(Summe,10); WriteLn
    END P;

  BEGIN
    A:=1; B:=2; P;
    WriteCard(A,10); WriteLn;
    WriteCard(B,10); WriteLn;
    WriteCard(Summe,10); WriteLn
  END LokalTest.
```

5.4 Formale Parameter

Wenn einer Prozedur bestimmte Größen zur Verarbeitung mitgeteilt werden sollen, so kann das natürlich über globale Größen geschehen. Dieser Weg – er wird üblicherweise mit «Seiteneffekt» bezeichnet – birgt eine große Gefahr in sich. Die globalen Größen werden instabil, ohne daß dieser Sachverhalt offen zutage tritt. Es bedarf einer genauen Analyse und sorgfältigen Dokumentation, um aufzuzeigen, wo welche Größen von einer Prozedur benötigt und verändert werden.

Ein zweiter Grund, auf Seiteneffekte wann immer möglich zu verzichten, liegt darin, daß nur so universell einsetzbare Prozeduren geschaffen werden können, die nicht an den Deklarationsteil eines übergeordneten Blocks gekoppelt sind.

Aus diesem Grund ist es möglich, den Datenaustausch zwischen einer Prozedur und deren Umgebung auf eine einzige Stelle zu konzentrieren, was der Programmsicherheit ganz unmittelbar zugute kommt. Diese Schnittstelle wird mit den formalen Parametern verwirklicht.

Formale-Parameterliste::="(" [Formalparameter]
{";" Formalparameter} ")".
Formalparameter::=["VAR"] Parametername
{"," Parametername} ":" Typ.

Beispiele:

```
PROCEDURE BerechneSumme(Zahl1, Zahl2 : CARDINAL;
VAR Summe : CARDINAL);

PROCEDURE SchreibeZeichen(Zeichen : CHAR);

PROCEDURE ZieheWurzel(Radikand : REAL;
VAR Wurzel : REAL);
```

Je nachdem, ob einem Formalparameter das Schlüsselwort «VAR» vorangeht oder nicht, unterscheidet man zwischen Referenz- und Wertparametern.

5.4.1 Wertparameter

Über Wertparameter werden einer Prozedur die Eingangsgrößen mitgeteilt. Der Parametername steht innerhalb der Prozedur für einen Wert des angegebenen Typs. Wird die Prozedur dann später aufgerufen, so wird anstelle des formalen Parameters ein Ausdruck übergeben. Ein bereits bekanntes Beispiel ist die Prozedur «WriteCard» aus dem Modul «InOut». Obwohl uns der genaue Aufbau dieser Prozedur nicht bekannt ist (und auch nicht weiter zu interessieren braucht), können wir die Parameterliste angeben:

```
PROCEDURE WriteCard(Wert, Stellen : CARDINAL); ....
```

Für «Wert» und «Stellen» kann beim Aufruf ein beliebiger Ausdruck vom Typ CARDINAL eingesetzt werden.

Innerhalb einer Prozedur können Wertparameter wie lokale Variable benutzt werden, die automatisch deklariert sind und beim Prozeduraufruf dadurch initialisiert werden, daß der übergebene Wert in die lokale Variable kopiert wird, z. B.:

Eine Prozedur mit dem Namen «xMalZeichen» soll ein Zeichen (CHAR) genau x-mal auf dem Bildschirm ausgeben und dann eine neue Zeile beginnen.

```
PROCEDURE xMalZeichen(X : CARDINAL; Zeichen : CHAR);
  VAR i : CARDINAL;
  BEGIN
    FOR i:=1 TO X DO Write(Zeichen) END;
    WriteLn
  END xMalZeichen;
```

Diese Prozedur könnte später dann beispielsweise so aufgerufen werden:

xMalZeichen(40,"X"); xMalZeichen(20+4-8,CHR(48+9)); ...

Daß Wertparameter wie normale Variable behandelt werden können, zeigt die folgende Abwandlung der obigen Prozedur:

```
PROCEDURE xMalZeichen(X : CARDINAL; Zeichen : CHAR);
  BEGIN
    WHILE X>0 DO
      Write(Zeichen);
      DEC(X)
    END; (* WHILE *)
    WriteLn
  END xMalZeichen;
```

Als nächstes Beispiel wollen wir eine Prozedur schreiben, die die Division zweier natürlicher Zahlen auf n Stellen genau rechtsbündig in ein vorgegebenes Feld schreibt. Diese Prozedur erhält die Parameter «Dividend», «Divisor», «Feld» und «Stellen». Wir gehen davon aus, daß die Prozeduren «Write», «WriteString» und «WriteCard» verfügbar sind.

Den Algorithmus für die Ausgabe leiten wir aus den Vorschriften ab, die wir selbst bei der schriftlichen Teilung zweier Zahlen ausführen:

Der Vorkommateil der Division ergibt sich als: Dividend DIV Divisor.

Die Stellenzahl für den Vorkommateil ist Feld-Stellen-1.
Schreibe den Dezimalpunkt.
Der Rest ist Dividend MOD Divisor.
Wiederhole Stellen-mal.
Schreibe Rest*10 DIV Divisor.
Rest ist Rest*10 MOD Divisor.

```
PROCEDURE SchreibeDivision(Dividend, Divisor, Feld, Stellen);
  VAR Rest : CARDINAL;
  BEGIN
    IF Divisor=0
    THEN WriteString("Fehler: Division durch 0!")
    ELSE
      WriteCard(Dividend DIV Divisor,Feld-Stellen-1);
      Rest:=Dividend MOD Divisor;
      Write(".");
      WHILE Stellen>0 DO
        WriteCard(10*Rest DIV Divisor,1);
        Rest:=10*Rest MOD Divisor;
        DEC(Stellen)
      END (* WHILE *)
    END (* IF *)
  END SchreibeDivision;
```

Als abschließendes Beispiel zu den Wertparametern wollen wir eine Prozedur betrachten, mit der der Cursor auf dem Bildschirm beliebig positioniert werden kann. Solche Terminal-Aktivitäten werden normalerweise über sogenannte ESCAPE-Sequenzen realisiert. Das heißt, es werden zunächst das Steuerzeichen <ESC> (33C), dann ein spezielles Steuerzeichen und schließlich die eigentlichen Koordinaten (eventuell in verschlüsselter Form) eingegeben. Die ESCAPE-Sequenz zur Bildschirmpositionierung lautet bei vielen Terminals: <ESC> «=» CHR(x+32) CHR(y+32)

```
PROCEDURE gotoxy(x,y : CARDINAL);
  BEGIN
    Write(33C); Write("="); Write(CHR(x+32)); Write(CHR(y+32))
  END gotoxy;
```

Diese unscheinbare Prozedur ist die Grundlage für jede vernünftige Bildschirmgestaltung.

5.4.2 Referenzparameter

Mit Wertparametern können Daten von außen in eine Prozedur gebracht werden. Das Prozedurkonzept wäre aber unvollständig, gäbe es nur diese eine Richtung der Datenübergabe. Der bidirektio-

nale (in zwei Richtungen) Datenaustausch wird durch sogenannte Referenzparameter realisiert. In der Parameterliste werden Referenzparameter durch Voranstellen des Schlüsselwortes «VAR» gekennzeichnet. Es weist darauf hin, daß beim Prozeduraufruf an dieser Stelle ausschließlich eine Variable übergeben werden darf, nicht jedoch ein Ausdruck oder eine Konstante.

Innerhalb der Prozedur weist zunächst nichts auf einen Unterschied zu Wertparametern hin. Da aber beim Prozeduraufruf keine Kopie gefertigt, sondern wirklich die übergebene Variable verwendet wird, kann eine Prozedur über Referenzparameter Ergebnisse nach außen mitteilen. Jede Änderung, die an dem übergebenen Parameter vorgenommen wird, wirkt unmittelbar auf die übergebene Variable.

Die folgende Prozedur berechnet die Summe zweier Zahlen. Die beiden Zahlen, «Summand1» und «Summand2», sind die Eingangsgrößen und deshalb als Wertparameter anzugeben. Das Ergebnis der Addition wird über den Referenzparameter «Summe» nach außen mitgeteilt.

```
PROCEDURE Summe(Summand1, Summand2 : CARDINAL; VAR Ergebnis : CARDINAL);
  BEGIN
    Ergebnis:=Summand1+Summand2
  END Summe;
```

Sind die Variablen X, Y und Z als Typ «CARDINAL» deklariert, so sind folgende Prozeduraufrufe zulässig:

```
Summe(1,30,X);
Summe(8 DIV Y,14-Z,X);
Summe(X,X,X); (* Hier hat X nach dem Aufruf den Wert 2*X *)
```

Verboten hingegen sind alle Aufrufe, bei denen als dritter Parameter keine CARDINAL-Variable übergeben wird:

```
Summe(1,2,3);
Summe(12-X,20,2*Z);
Summe(X,Y,Y+Z);
```

Im nächsten Beispiel wollen wir von der Tastatur eine Zahl einlesen, die innerhalb der Grenzen «von» und «bis» liegt:

```
PROCEDURE LiesZahl(VAR Zahl : CARDINAL; von, bis : CARDINAL);
  BEGIN
    REPEAT
      WriteString("Bitte geben Sie eine Zahl zwischen ");
      WriteCard(von,1);
      WriteString(" und ");
      WriteCard(bis,1);
      WriteString(" ein: ");
      ReadCard(Zahl);
      WriteLn
    UNTIL (Zahl>=von) AND (Zahl<=bis)
  END LiesZahl;
```

Die folgende Prozedur berechnet die Fakultät einer eingegebenen Zahl:

```
PROCEDURE FakultaetsBerechnung(Zahl : CARDINAL; VAR Fakultaet : CARDINAL);
  BEGIN
    Fakultaet:=1;
    WHILE Zahl>0 DO
      Fakultaet:=Fakultaet*Zahl;
      DEC(Zahl)  (* oder Zahl:=Zahl-1 *)
    END (* WHILE *)
  END FakultaetsBerechnung;
```

Zusammenfassend können folgende Punkte zu den Parametern herausgestellt werden:

- ☐ Wertparameter sind ausschließlich Eingangsgrößen einer Prozedur.
- ☐ Referenzparameter erlauben den Datenaustausch in beiden Richtungen.
- ☐ Die Parameterliste einer Prozedur kann beliebig lang sein.
- ☐ Beim Prozeduraufruf muß die Reihenfolge der aktuellen Parameter mit der der formalen Parameter übereinstimmen.

Um eine möglichst hohe Programmsicherheit zu gewährleisten, sollten nur dann Referenzparameter verwendet werden, wenn über sie wirklich ein Ergebnis nach außen mitgeteilt wird. Leider findet man in der «Fortgeschrittenen-Literatur» immer wieder Hinweise und Tips zur Programmoptimierung (in bezug auf Ablaufgeschwindigkeit und Speicherbedarf), die besagen, Referenzparameter auch dann einzusetzen, wenn sie nur die Funktion von Wertparametern übernehmen, also nur Eingangsgrößen sind. Solchen Tips ist an dieser Stelle leidenschaftlich zu widersprechen. Die modernen Computer verbinden hohe Leistungsfähigkeit und große Speicherkapazitäten in einem Maße, daß die geringfügigen Effizienzgewinne in keinem Verhältnis zur gewonnenen Sicherheit stehen.

Die (meist kaum meßbaren) Effizienzgewinne kommen übrigens daher, daß bei Referenzparametern keine Kopie an eine lokale Variable erfolgen muß.

Aufgaben:

1. Im Deklarationsteil eines Programm-Moduls befindet sich folgende Prozedurbeschreibung:

   PROCEDURE xyz(VAR A,B : CARDINAL; C : CARDINAL); ...

   Welche der folgenden Prozeduraufrufe sind zulässig (VAR n,m : CARDINAL)?

   xyz(n,m,3-9);
   xyz(n,n,m);
   xyz(n,1000 DIV m,m);
   xyz(n,m,2*m-n+ORD("A"))

2. Welche Parameterarten haben normalerweise Ausgabeprozeduren, und welche haben Eingabeprozeduren?

5.5 Die RETURN-Anweisung

Eine Prozedur wird – genauso wie ein Programm – immer vollständig abgearbeitet. In manchen Fällen kann es nötig sein (etwa beim Auftreten von irgendwelchen Fehlern), eine Prozedur vorzeitig zu verlassen. Hier kann dann die RETURN-Anweisung eingesetzt werden, die nur aus dem Schlüsselwort «RETURN» besteht.

RETURN-Anweisung::="RETURN".

Die Ausführung der RETURN-Anweisung bewirkt einen Sprung an das Ende des Anweisungsteiles der Prozedur, in der die Anweisung steht. Innerhalb eines Anweisungsteiles einer Prozedur darf die RETURN-Anweisung beliebig oft vorkommen.

Im Anweisungsteil eines Moduls darf die RETURN-Anweisung nicht benutzt werden. Sie ist somit in ähnlicher Weise an Prozeduren gekoppelt wie EXIT an die LOOP-Anweisung.

Als Beispiel dient eine Prozedur, die eine CARDINAL-Zahl von der Tastatur einliest. Tritt dabei ein Fehler auf, so wird die Proze-

dur mit einem Fehlercode verlassen. Die Fehlercodes sind dabei wie folgt festgelegt:

0 : Alles in Ordnung
1 : Illegale Zeichen in der Zahl
2 : CARDINAL-Überlauf (Zahl ist zu groß)

Die Prozedur geht davon aus, daß die Konstante «EOL» bekannt ist, die das Ende einer Eingabezeile angibt. «EOL» (End Of Line = Ende der Zeile) ist von Typ «CHAR» und wird normalerweise vom Modul «InOut» exportiert, in manchen Systemen vom Modul «ASCII».

```
PROCEDURE LiesCard(VAR Zahl, FehlerCode : CARDINAL);
  VAR Zeichen : CHAR;
  BEGIN
    Zahl:=0; FehlerCode:=0;
    LOOP
      Read(Zeichen);
      CASE Zeichen OF
         EOL : RETURN
       | '0'..'9' : IF Zahl<=(MAX(CARDINAL)-(ORD(Zeichen)-ORD('0'))) DIV 10
                    THEN Zahl:=10*Zahl+(ORD(Zeichen)-ORD('0')
                    ELSE FehlerCode:=2; RETURN
                    END
      ELSE FehlerCode:=1; RETURN
      END (* CASE *)
    END (* LOOP *)
  END LiesCard;
```

Die in der Prozedur vorkommenden Formeln (Ausdrücke) setzen sich folgendermaßen zusammen: Wenn eine Zahl von links nach rechts ziffernweise eingegeben wird, so berechnet sich ihr Wert aus 10 * Zahl + Ziffer.

Beispiel: Eingegeben wird die Ziffernfolge 1, 2 und 3.

Start: Zahl:=0

Gelesene Ziffer	1	2	3
Zahl	0*0+1=1	1*10+2=12	12*10+3=123

Um in Modula-2 den (Zahlen-)Wert einer Ziffer zu bestimmen, wird von ihrem ASCII-Code derjenige der Ziffer 0 abgezogen (ORD(Zeichen)-ORD('0')). Der Überlauftest kann nun nicht durch folgende Konstruktion erfolgen:

```
IF Zahl*10 + (ORD(Zeichen)-ORD('0')) <= MAX(CARDINAL)
THEN ...
```

Hier würde ein eventueller Überlauf auftreten, bevor (!) er erkannt werden könnte. Aus diesem Grund wird dieser Ausdruck folgendermaßen umgeformt:

```
Zahl*10   <= MAX(CARDINAL) - (ORD(Zeichen)-ORD('0'))
Zahl      <= (MAX(CARDINAL) - (ORD(Zeichen)-ORD('0')))
             DIV 10
```

Der Einsatz der RETURN-Anweisung ist nur bei komplexen oder besonders zeitkritischen Prozeduren angezeigt. Im weiteren Verlauf dieses Kurses werden Sie den Einsatz der RETURN-Anweisung noch mehrfach kennenlernen. Das obige Beispiel kann, ohne an Klarheit zu verlieren, auch mit einer REPEAT-Anweisung realisiert werden.

```
PROCEDURE LiesCard(VAR Zahl, FehlerCode : CARDINAL);
  VAR Zeichen : CHAR;
  BEGIN
    Zahl:=0; FehlerCode:=0;
    REPEAT
      Read(Zeichen);
      CASE Zeichen OF
         EOL : (* nichts *)
       | '0'..'9' : IF Zahl<=(MAX(CARDINAL)-(ORD(Zeichen)-ORD('0'))) DIV 10
                    THEN Zahl:=10*Zahl+(ORD(Zeichen)-ORD('0')
                    ELSE FehlerCode:=2
                    END
      ELSE FehlerCode:=1
      END (* CASE *)
    UNTIL (Zeichen=EOL) OR (FehlerCode>0)
  END LiesCard;
```

Aufgaben:

1. Die Prozedur «LiesCard» hat zwei kleine Schönheitsfehler. Zum einen werden führende Leerzeichen nicht übersprungen, zum anderen wird eine Leerzeile als Eingabe der Zahl 0 fehlinterpretiert. Beheben Sie diese Fehler.
2. Schreiben Sie eine Prozedur «LiesInt» in gleicher Weise wie «LiesCard».

5.6 Funktionsprozeduren

Der Sonderfall einer Prozedur liegt dann vor, wenn sie genau einen Rückgabewert liefert, alle anderen Parameter ausschließlich Eingangsgrößen sind. Handelt es sich bei dem Rückgabewert um einen der Grundtypen (einschließlich REAL), so können derartige Prozeduren als sogenannte «Funktionsprozeduren» formuliert werden. Dabei wird der Rückgabewert nicht in die Parameterliste aufgenommen. Vielmehr repräsentiert der Prozedurname dann selbst diesen Rückgabewert, dessen Typ an die Parameterliste – durch Doppelpunkt abgetrennt – angehängt wird.

Funktionsprozedur::="PROCEDURE" "(" [FormalParameter]
{";" FormalParameter} ")"
":" FunktionsTyp.

Funktionsprozeduren entsprechen genau dem, was wir bisher unter dem Namen Funktion kennengelernt haben. Wichtig ist – im Unterschied zu normalen Prozeduren – daß eine Funktionsprozedur immer eine Parameterliste haben muß, die jedoch auch leer sein kann. Zudem muß jede Funktionsprozedur mindestens eine RETURN-Anweisung haben, wodurch der Funktionswert nach außen gegeben wird. In Funktionsprozeduren hat die RETURN-Anweisung folgende Gestalt:

RETURN-Anweisung::="RETURN" Ausdruck.

Der Ausdruck muß vom Typ der Funktionsprozedur sein. Da der Name einer Funktionsprozedur den Funktionswert repräsentiert, muß der Aufruf innerhalb eines Ausdrucks erfolgen. Man kann auch sagen, eine Funktionsprozedur wird durch Verwendung aufgerufen.

Beispiele:

```
PROCEDURE Fakultaet(N : CARDINAL):CARDINAL;
  VAR i, Fak : CARDINAL;
  BEGIN
    Fak:=1;
    FOR i:=1 TO N DO Fak:=Fak*i END;
    RETURN Fak
  END Fakultaet;
```

Ein Gebrauch dieser Funktion könnte beispielsweise so lauten:

```
   WriteCard(Fakultaet(8),10);

PROCEDURE Klein(Zeichen) : CHAR;
 (* Gegenstück von 'CAP', wandelt Groß- in Kleinbuchstaben *)
  BEGIN
    IF (C>='A') AND (C<='Z') (* Großbuchstabe *)
    THEN RETURN CHR(ORD(Zeichen)+ORD('a')-ORD('A'))
    ELSE RETURN Zeichen
    END
  END Klein;
```

Folgende Punkte sollten im Sinne eines guten Programmierstils beim Einsatz von Funktionsprozeduren unbedingt beachtet werden:

- ☐ Die Parameterliste einer Funktionsprozedur sollte ausschließlich aus Wertparametern bestehen.
- ☐ Eine Funktionsprozedur sollte niemals irgendwelche Seiteneffekte haben.

Aufgaben:

1. Schreiben Sie eine Funktionsprozedur «CARD», die den CARDINAL-Wert einer Ziffer liefert, also beispielsweise CARD('5') –> 5.
2. Schreiben Sie eine Funktionsprozedur, die angibt, ob eine übergebene CARDINAL-Zahl gerade (ohne Rest durch 2 teilbar) ist.
3. Schreiben Sie eine Funktionsprozedur, die prüft, ob es sich bei der übergebenen CARDINAL-Zahl um eine Primzahl handelt.
4. Schreiben Sie eine Funktionsprozedur «FRAC», die den Nachkommateil einer übergebenen REAL-Zahl liefert.

6 Module

Mit Prozeduren kann ein komplexes Problem in einzelne Teilaufgaben zerlegt werden. Dies ermöglicht eine einfache, übersichtliche und sichere Programmierung. Trotzdem reicht das Prozedurkonzept für moderne Programmiertechniken nicht aus. Um dieser Behauptung zustimmen zu können, müssen wir die Grenzen der Prozeduren genauer untersuchen.

Als erstes ist festzustellen, daß Prozeduren prinzipiell an das Programm-Modul gebunden sind, in dessen Deklarationsteil sie beschrieben wurden. Wird in einem anderen Programm dieselbe Prozedur benötigt, so muß sie dort erneut gleichlautend geschrieben werden. Moderne Textverarbeitungssysteme unterstützen zwar in hohem Maß solche Texteinbindungen, doch bleibt es ärgerlich und überflüssig, daß derart eingebundene Textteile erneut übersetzt werden müssen, obwohl ganz offensichtlich eine einmalige Übersetzung ausreichen würde. Hier ist eine Art Bibliothek wünschenswert, aus der man bei Bedarf die bereits fix und fertig übersetzten Bausteine entnehmen kann.

Ein anderes Problem tritt auf, wenn ein Programm so groß ist, daß seine Realisierung auf mehrere Programmierer verteilt wird. Hier ist es nicht nur wünschenswert, sondern sogar unabdingbar, daß in jedem der derart entstehenden Programmteile alle Details, die nicht unmittelbar an das Gesamtprogramm weitergegeben werden, vor den anderen Teilnehmern versteckt werden können. Bei diesem «Verstecken» geht es gar nicht darum, die übrigen Programmierer vom eigenen Know-how auszuschließen. Vielmehr besteht die Gefahr, daß fremde Strukturen ohne böse Absicht mißbräuchlich genutzt werden, was zu äußerst schwer entdeckbaren Programmfehlern führen kann. Ein wesentlich sichererer Weg besteht darin, nur die gewünschten Bezeichner als für andere zugänglich auszuzeichnen, also eine Sichtbarkeitshülle um ganze Programmteile zu legen und damit das Innere vor fremden Zugriffen zu schützen.

Aber auch innerhalb eines Programm-Moduls kann es nötig sein, einen einzelnen Komplex aus mehreren Prozeduren zu schützen, wenn diese Prozeduren beispielsweise via Seiteneffekt einen gemeinsamen Datenbestand bearbeiten und dieser Datenbestand nicht unmittelbar zum Hauptprogramm gehört. Zudem wäre es sinnvoll, wenn solche Komplexe einen eigenen Anweisungsteil hätten, in dem alle benötigten Initialisierungen vorgenommen würden, ohne daß sich das Hauptprogramm darum kümmern muß. Auf diese Weise kann verhindert werden, daß die u.U. notwendige Definition eines Anfangszustandes vergessen wird.

Alle diese Forderungen werden von Modula-2 durch die Bereitstellung des Modul-Konzepts erfüllt. Ein Modul bildet eine in jeder Richtung undurchlässige Hülle. Im Gegensatz zu Prozeduren, wo innerhalb eines Blocks auch alle äußeren Blöcke bekannt sind, ist ein Modul vollkommen isoliert. Einzig durch ausdrückliche Erklärung kann für einzelne Bezeichner die Isolation aufgehoben werden.

6.1 Programm-Module

Bisher haben wir es ausschließlich (mit Ausnahme des Vorgriffs «BooleanInOut») mit Programm-Moduln zu tun gehabt. Diese erzeugen – wie an den vielen Beispielen gesehen – ablauffähige Programme. Sie können keine Bezeichner nach außen abgeben (exportieren). Um Bezeichner aus Bibliotheksmoduln innerhalb eines Programm-Moduls sichtbar zu machen, müssen sie explizit importiert werden. Zur Wiederholung noch einmal die Syntax des Programm-Moduls:

Programm-Modul::="MODULE" Modulname ";"
[Import-Listen] Block Modulname ".".

Ein Programm-Modul ist eine sogenannte Compilationseinheit. Das bedeutet, daß es vom Compiler vollständig bearbeitet werden kann.

6.2 Bibliotheksmodule

Mit den Bibliotheksmoduln ist es möglich, Konstante, Typen, Variable und Prozeduren beliebig verfügbar zu machen. Um dem gewünschten Verbergungsprinzip zu genügen, bestehen Bibliotheksmodule aus zwei Teilen, dem Definitions- und dem Implementationsteil.

6.2.1 Das Definitionsmodul

Im Definitionsmodul werden alle Bezeichner aufgeführt, die vom Bibliotheksmodul bereitgestellt (exportiert) werden.

Definitionsmodul::="DEFINITION" "MODULE" Modulname ";"
[Import-Listen] Block Modulname ".".

Syntaktisch unterscheidet sich ein Definitionsmodul nur im Schlüsselwort «DEFINITION» von einen Programm-Modul. Dennoch gilt es einige Besonderheiten zu beachten:

- Der Block eines Definitionsmoduls ist immer leer, besteht also nur aus dem Schlüsselwort «END».
- Bei den Prozeduren wird nur der Prozedurkopf angegeben.

Auch ein Definitionsmodul bildet eine Compilationseinheit. Es wird jedoch kein ausführbares Programm erzeugt. Dennoch kann nach der Übersetzung bereits auf die exportierten Bezeichner zugegriffen werden. Da die genaue Ausführung aber noch fehlt, können die exportierten Prozeduren noch nicht verwendet werden.

Hinweis: Diese Beschreibung folgt der neuen Fassung von Modula-2. In älteren Systemen müssen alle exportierten Bezeichner in einer eigenen Export-Liste aufgeführt werden.

Definitionsmodul::="DEFINITION" "MODULE" Modulname ";"
[Import-Listen]
Export-Liste Block Modulname ".".
Export-Liste::="EXPORT" "QUALIFIED" Bezeichner
{"," Bezeichner} ";".

Angenommen, wir wollen ein Bibliotheksmodul erstellen, das sogenannte Zufallszahlen liefert, die in allen Simulationsprogrammen benötigt werden. Unser Modul soll sowohl Zahlen von Typ CARDINAL als auch vom Typ REAL liefern. Während die REAL-Zahlen Werte von 0.0 (eingeschlossen) bis 1.0 (ausgeschlossen) annehmen sollen, soll bei den CARDINAL-Zahlen die gewünschte Obergrenze in Form eines Parameters angegeben werden.

```
DEFINITION MODULE Random;
(* Liefert CARDINAL- und REAL-Zufallszahlen *)

PROCEDURE RandomReal():REAL;
(* 0.0<= RandomReal < 1.0 *)

PROCEDURE RandomCard(Maximum : CARDINAL):
CARDINAL;

(* 0 <= RandomCard < Maximum *)

END Random.
```

Definitionsmodule erfüllen eine Doppelfunktion. Ihre übersetzte Form stellt dem Modula-2-System die Schnittstelle zum Bibliotheksmodul zur Verfügung. Mit den darin enthaltenen Funktionen kann der Compiler später prüfen, ob die exportierten Bezeichner korrekt angewandt werden. Die Textform eines Definitionsmoduls ist für alle Programmierer von Interesse, die an den Leistungen des Moduls teilhaben wollen. Aus diesem Grund bedürfen Definitionsmodule immer einer ausführlichen Dokumentation. Die Dokumentation umfaßt oft mehrere Seiten, Ergebnisse und Arbeitsweise der exportierten Bezeichner sind hier genau darzulegen.

6.2.2 Das Implementations-Modul

Im Implementationsteil schließlich folgt die Ausführung eines Bibliotheksmoduls. Innerhalb eines Implementations-Moduls kann der zugehörige Definitionsteil als bekannt vorausgesetzt werden.

```
Implementations-Modul::="IMPLEMENTATION" "MODULE"
                         Modulname ";"
                         [Import-Listen] Block Modulname ".".
```

Hier ist die Ähnlichkeit zum Programm-Modul sogar noch stärker. Folgende Besonderheiten sind zu beachten:

- Alle Prozeduren des dazugehörigen Definitionsmoduls müssen hier ausformuliert werden. Die Prozedurköpfe müssen gleichlautend übernommen werden.
- Alle weiteren Deklarationen sind lokal zum Modul.
- Ein Implementations-Modul kann einen nicht-leeren Anweisungsteil enthalten. Dieser wird ausgeführt, bevor zum ersten Mal auf einen exportierten Bezeichner zugegriffen wird. Dadurch ist es möglich, Objekte des Moduls zu initialisieren.

Zum besseren Verständnis nun der Implementationsteil unseres Zufallsmoduls:

```
IMPLEMENTATION MODULE Random;

  (* Bestimmt eine Folge von Zufallszahlen nach dem HP(Hewlett-Packard)-
     Verfahren. *)

  FROM InOut IMPORT ReadCard, WriteString, WriteLn, Done;

  CONST irrational = 1.1415926536;
  (* HP wählt hier die Kreiszahl pi, was aber auf 16-Bit-Systemen zu
     überläufen führt. Deshalb nehmen wir pi-2.0. *)

  VAR Startwert : CARDINAL;
      Zufallszahl : REAL;

  PROCEDURE NeueZahl;

    PROCEDURE FRAC(Zahl : REAL):REAL;
    (* Berechnet den Nachkommateil einer reellen Zahl *)
      BEGIN
        RETURN Zahl-FLOAT(TRUNC(Zahl))
      END FRAC;

    PROCEDURE HochAcht(Zahl : REAL):REAL;
    (* Berechnet Zahl hoch 8 *)
      VAR i : CARDINAL;
      BEGIN
        FOR i:=1 TO 3 DO Zahl:=Zahl*Zahl END;
        RETURN Zahl
      END HochAcht;

    BEGIN (* NeueZahl *)
      Zufallszahl:=FRAC(HochAcht(Zufallszahl+irrational))
    END NeueZahl;

  PROCEDURE RandomReal():REAL;
    BEGIN
      NeueZahl;
      RETURN Zufallszahl
    END RandomReal;
```

```
PROCEDURE RandomCard(bis : CARDINAL):CARDINAL;
  BEGIN
    NeueZahl;
    RETURN TRUNC(FLOAT(bis)*Zufallszahl)
  END RandomCard;

BEGIN (* Initialisierung des Zufallsgenerators *)
  REPEAT
    WriteString("Startwert für Zufallsgenerator (Zahl > 0) ");
    ReadCard(Startwert);
    WriteLn
  UNTIL Done AND (Startwert>0);
  Zufallszahl:=1.0/FLOAT(Startwert)
END Random.
```

Dieses Modul kann erst übersetzt werden, wenn das zugehörige (und gleichnamige) Definitionsmodul erfolgreich übersetzt wurde. Im Anschluß daran steht ein vollständiges Bibliotheksmodul «Random» zur Verfügung. Ein kleines Testprogramm zeigt die Funktionsweise.

```
MODULE RandomTest;

  FROM Random IMPORT RandomCard;
  FROM InOut IMPORT WriteCard, WriteLn;

  VAR i : CARDINAL;

  BEGIN
    FOR i:=1 TO 15 DO
      WriteCard(RandomCard(100),10);
      WriteLn
    END (* FOR *)
  END RandomTest.
```

Probelauf:

```
Startwert für Zufallsgenerator (Zahl > 0) 2
73
84
88
11
91
42
53
53
28
20
33
```

```
43
35
67
4
```

Im weiteren Verlauf dieses Kurses werden noch mehrere Bibliotheksmodule vorgestellt.

6.2.3 Import aus Bibliotheksmoduln

Für den Zugriff auf Bibliotheksmodule gibt es zwei Möglichkeiten. Die erste besteht darin, in einer Importliste den Namen des Moduls sowie alle gewünschten Bezeichner anzugeben:

"FROM" Modulname "IMPORT" Bezeichner {"," Bezeichner} ";"

Dieses Verfahren ist dann nicht mehr anzuwenden, wenn verschiedene Bibliotheksmodule gleiche Bezeichner liefern. Dafür kann ein gesamtes Bibliotheksmodul importiert werden:

"IMPORT" Modulname ";"

Der Zugriff auf die einzelnen Objekte erfolgt dadurch, daß dem jeweiligen Bezeichner der Modulname und ein Punkt vorangestellt werden. Auf diese Weise werden Namenskonflikte vermieden (qualifizierter Import).

Beispiel:

```
IMPORT InOut;
FROM Terminal IMPORT WriteString, WriteLn;
...
InOut.WriteString("Hier schreibt 'InOut'");
WriteString("Und hier 'Terminal'")
```

Die beiden Importarten können beliebig gemischt werden.

6.3 Lokale Module

Module können aber auch innerhalb des Deklarationsteil eines Blocks auftreten. Man spricht dann von einem lokalen Modul. Es ist nicht getrennt von seiner Umgebung übersetzbar.

Deklaration::={Konstantenvereinbarung|Typendefinition|
Variablendeklaration|Prozedurbeschreibung|
Lokales-Modul}.

Lokales-Modul::="MODULE" Modulname ";" [Import-Listen]
[Export-Liste]
Block Modulname ";".
Export-Liste::="EXPORT" ["QUALIFIED"] Bezeichner
{"," Bezeichner} ";".

Folgende Besonderheiten sind festzuhalten:

- Ein lokales Modul endet mit einem Semikolon anstelle des sonst üblichen Punktes.
- Ein lokales Modul ohne Exportliste ist zwar möglich, aber kaum sinnvoll.
- Es kann ausschließlich Bezeichner aus seiner Umgebung importieren. Aus diesem Grund fällt die Konstruktion «FROM Modulname» bei den Importlisten weg.

Je nachdem, ob das Schlüsselwort «QUALIFIED» vorhanden ist oder nicht, unterscheidet man zwischen qualifiziertem und nicht-qualifiziertem Export.

Beim qualifizierten Export wird auf die exportierten Objekte zugegriffen, indem der Modulname gefolgt von einem Punkt dem jeweiligen Bezeichner vorangestellt wird. Dadurch werden Namenskollisionen automatisch vermieden. Andernfalls werden die aufgeführten Bezeichner unmittelbar zur Verfügung gestellt, das bedeutet, daß Modulname und Punkt wegfallen dürfen (jedoch nicht müssen!).

Beispiel:

```
MODULE LokalModulTest;

  FROM InOut IMPORT WriteString, WriteLn;

   MODULE LokalesModul1;
     IMPORT WriteString, WriteLn;
```

```
    EXPORT Meldung;
    PROCEDURE Meldung;
      BEGIN
        WriteString("Hier ist das lokale Modul Nr.1");
        WriteLn
      END Meldung;
    END LokalesModul1;

  MODULE LokalesModul2;
    IMPORT WriteString, WriteLn;
    EXPORT QUALIFIED Meldung;
    PROCEDURE Meldung;
      BEGIN
        WriteString("Hier ist das lokale Modul Nr.2");
        WriteLn
      END Meldung;
    END LokalesModul2;

BEGIN
  Meldung;
  LokalesModul1.Meldung;
  LokalesModul2.Meldung;
END LokalModulTest.
```

Probelauf:

Hier ist das lokale Modul Nr.1
Hier ist das lokale Modul Nr.1
Hier ist das lokale Modul Nr.2

Da der Anweisungsteil eines lokalen Moduls automatisch vor der Programmausführung abgearbeitet wird, eignen sich lokale Module besonders für Initialisierungsaufgaben.

Beispiel:

Bei der Analyse von Texten ist es häufig nötig, ein Zeichen über das gerade gelesene vorauszuschauen. Dies kann am einfachsten dadurch erreicht werden, wenn ein Zeichen wieder in die Textdatei zurückgeschrieben werden kann. Hierfür eignet sich ein lokales Modul mit eigenem Gedächtnis in Form eines Puffers, der genau ein Zeichen zwischenspeichern kann. Hat dieser Puffer den Wert 0C, so ist er leer und kann bei Bedarf ein neues Zeichen aufnehmen.

Hier folgt ein kleines Programm, das mit Hilfe der gepufferten Eingabe den Quelltext eines Modula-2-Programms so kopiert, daß in der Kopie alle Kommentare entfernt sind. Das Vorausschauen wird benötigt, wenn das Zeichen (gelesen wird. Ist das nächste Zeichen *, so beginnt ein Kommentar. Ebenfalls eine Vorausschau

wird beim Kommentarende benutzt, das durch die Zeichenfolge *) angezeigt wird.

Der Tatsache, daß in Modula-2 Kommentare beliebig verschachtelt sein dürfen, tragen wir durch das Mitzählen der Kommentartiefe Rechnung. Hat diese den Wert Null, so liegt kein Kommentar vor, und das Zeichen kann ausgegeben werden.

```
MODULE KommentarEntfernen;

  FROM InOut IMPORT Read, Write, WriteString, WriteLn, Done,
                    OpenInput, OpenOutput, CloseInput, CloseOutput;

  VAR Zeichen, Naechstes : CHAR;
      KommentarTiefe : CARDINAL;

  MODULE GepuffertesLesen; (* Lokales Modul *)

    IMPORT Read;  (* Ist nicht automatisch bekannt! *)
    EXPORT ReadChar, PushBack;

    VAR Puffer : CHAR;

    PROCEDURE ReadChar(VAR C : CHAR);
    (* Falls der Puffer leer ist, wird ein Zeichen mit Read gelesen.
       Ansonsten wird das Zeichen des Puffers übergeben *)
      BEGIN
        IF Puffer=0C
        THEN Read(C)
        ELSE C:=Puffer; Puffer:=0C
        END
      END ReadChar;

    PROCEDURE PushBack(C : CHAR);
    (* Legt das Zeichen C im Puffer ab *)
      BEGIN
        Puffer:=C
      END PushBack;

    BEGIN  (* Initialisierungsteil von GepuffertesLesen *)
      Puffer:=0C  (* Puffer ist leer *)
    END GepuffertesLesen;

  BEGIN (* KommentarEntfernen *)
   WriteString("Entfernen der Kommentare in einem Programmtext");
   WriteLn; WriteLn;
   OpenInput("MOD");
   IF NOT Done THEN WriteString("Keine Datei"); HALT END;
   OpenOutput("");
   KommentarTiefe:=0;
   ReadChar(Zeichen);
   WHILE Done DO
     IF Zeichen="("                (* Test auf Kommentaranfang *)
     THEN
       ReadChar(Naechstes);
       IF Naechstes='*'
       THEN INC(Kommentartiefe)
       ELSE PushBack(Naechstes)
```

```
      END
    ELSIF Zeichen="*"          (* Test auf Kommentarende *)
    THEN
      ReadChar(Naechstes);
      IF Naechstes=")"
      THEN DEC(KommentarTiefe)
      ELSE PushBack(Naechstes)
      END
    END; (* IF *)
    IF KommentarTiefe=0 THEN Write(Zeichen) END;
   ReadChar(Zeichen)
  END; (* WHILE *)
  CloseOutput;
  CloseInput
END KommentarEntfernen.
```

Aufgaben:

1. Studieren Sie die Unterlagen zu Ihrem Computerterminal, und schreiben Sie ein Bibliotheksmodul «Video», das folgende Routinen (Prozeduren) liefert:

ClrScr	– löscht den Bildschirm
gotoxy(X,Y : CARDINAL)	– setzt den Cursor (Schreibmarke) auf Spalte X und Zeile Y
EraEol	– löscht von der aktuellen Cursorposition bis zum Ende der Zeile
EraEos	– löscht von der aktuellen Cursorposition bis zum Rest des Bildschirms

2. Das Programm «KommentarEntfernen» kann in eine Falle laufen, wenn die Zeichenfolgen (* oder *) in Zeichenketten auftreten. In einem solchen Fall wird irrtümlich die Kommentartiefe herauf- bzw. herabgesetzt. Ist es möglich, diesen Fehler mit wenig Aufwand zu beheben?

7 Selbstdefinierte Typen

Aufbauend auf den Grundtypen, können in Modula-2 weitere Datentypen definiert werden. Dazu gibt es zwei Möglichkeiten:

1. Die Typdefinition steht direkt bei der Variablen-Deklaration. Dann hat dieser Typ keinen eigenen Namen und heißt deshalb «anonym».
2. Ein neuer Typ mit Angabe seines Namens wird definiert. Nach der Definition kann auf diesen Typ – unter Verwendung seines Namens – zugegriffen werden.

Typdefinition::="TYPE" {TypName "=" Typ ";"}.
Typ ::= Grundtyp | Aufzählungstyp | Unterbereichstyp | Mengentyp | Feldtyp | Rekordtyp | Prozedurtyp | Zeigertyp.

7.1 Aufzählungstyp

Im einfachsten Fall wird ein neuer Typ dadurch geschaffen, daß sein Wertebereich vollständig aufgezählt wird.

Aufzählungstyp::="(" Bezeichner {"," Bezeichner} ")".

Wenn wir beispielsweise einen Typ «Farbe» benötigen, so kann das mit folgender Typdefinition geschehen:

TYPE Farbe = (rot, gruen, blau, gelb, schwarz, weiss);

Mit dieser Definition wird nicht nur der Datentyp «Farbe» geschaffen. Gleichzeitig werden die Konstanten dieses Typs (rot, gruen, ...) eingeführt. Mit der Reihenfolge der Aufzählung wird die Ordnung dieser Konstanten festgelegt. Es gilt also:

rot < gruen < blau < gelb < schwarz < weiss

Innerhalb dieser Ordnung gilt die Nachfolger- und Vorgängerbeziehung. Das bedeutet, daß «gruen» der Nachfolger von «rot» und der Vorgänger von «blau» ist.

Der neu geschaffene Typ ist also genauso angeordnet wie die natürlichen Zahlen. Aus diesem Grund ist es auch möglich, die Ordnungszahl einer Konstanten zu bestimmen. Dabei ist die Ordnungszahl der ersten Konstanten 0, die der zweiten 1 usw.

Mit der Standardfunktion «ORD» kann man Auskunft über die Ordnungszahl erhalten.

```
ORD(rot)    -> 0
ORD(gelb)   -> 3
```

Mit der obigen Typdefinition wurde also ein neuer Datentyp geschaffen, der sich wie die aufzählbaren Grundtypen CARDINAL, INTEGER, CHAR und BOOLEAN verhält. Auf ihn sind auch die Standardprozeduren INC und DEC anwendbar.

Weitere Beispiele für Aufzählungstypen:

```
TYPE    Wahrheitswert = (wahr, falsch);
        Computerzubehoer = (Drucker, Plotter, Monitor, Maus,
                            Joystick, Tastatur);
        DruckerStatus = (Busy, Ready, OutOfPaper);
        Vorspeisen = (Suppe, Salat, Schnecken);
```

7.1.1 Eine Ampelsteuerung

Als erstes Beispiel wollen wir eine einfache Verkehrsampel simulieren. Da Konstante von Aufzählungstypen (wie auch von BOOLEAN) nicht direkt ein- und ausgegeben werden können, schreiben wir uns selbst eine entsprechende Ausgabeprozedur.

```
MODULE Ampel;
  FROM InOut IMPORT WriteString, WriteLn;
  TYPE AmpelFarbe = (gruen, gelb, rot, rotgelb);
  VAR  Ampel : AmpelFarbe;

  PROCEDURE SchreibAmpelFarbe(Farbe : AmpelFarbe);
    BEGIN
      CASE Farbe OF
        gruen : WriteString("grün")
      | gelb  : WriteString("gelb")
      | rot   : WriteString("rot ")
      | rotgelb : WriteString("rot und gelb")
      END (* CASE *)
    END SchreibAmpelFarbe;
```

```
BEGIN (* Ampel *)
  LOOP
    FOR Ampel:=gruen TO rotgelb DO SchreibAmpelFarbe(Ampel); WriteLn END
  END (* LOOP *)
END Ampel.
```

Probelauf:

grün
gelb
rot
rot und gelb
grün
gelb
...

Sie sehen, daß auch selbstdefinierte Aufzählungstypen in CASE- oder FOR-Anweisungen verwendet werden können.

Aufgabe: Das Programm läuft in einer Endlosschleife. Ändern Sie es so ab, daß bei einem beliebigen Tastendruck die Schleife verlassen wird.

7.1.2 Ein endlicher Automat

Ein weites Feld der Computeranwendungen ist die Textverarbeitung und -analyse. Als etwas ausführlicheres Anwendungsbeispiel werden wir später ein Programm entwickeln, das einen Modula-2-Programmtext so bearbeitet, daß die Schlüsselwörter fettgedruckt und die Zeilen mit Nummern versehen werden. Dabei müssen jedoch Stringkonstante und Kommentare speziell behandelt werden, da hier vorkommende Schlüsselwörter normal ausgegeben werden sollen. Als Vorübung dazu dient das folgende Programm, das aus einem Modula-2-Text alle Stringkonstanten extrahiert.

Zur Lösung von Problemen dieser Art eignet sich die Methode der «endlichen Automaten». Dabei wird eine Maschine konstruiert, die eine bestimmte Anzahl verschiedener Zustände annehmen kann. Je nach Zusammentreffen von Eingabedaten und Zustand werden dann Aktionen ausgelöst und/oder der Zustand geändert.

Für unseren Automaten benötigen wir drei Zustände:

«ZeichenLesen», «String1Lesen» und «String2Lesen».

Der Zustand «ZeichenLesen» soll der normale sein. Wenn jedoch ein Hochkomma gelesen wird, so zeigt das an, daß im Text eine Stringkonstante beginnt. In diesem Fall schalten wir in den Zustand «String1Lesen» um. In diesem Zustand verbleibt unser Automat so lange, bis das nächste Hochkomma gelesen wird. Erst dann wird wieder in den Zustand «ZeichenLesen» umgeschaltet. Auf diese Weise werden im Zustand «String1Lesen» ganz automatisch auch Gänsefüßchen als Bestandteile der Stringkonstante erkannt. Analog verfahren wir, wenn ein doppeltes Anführungszeichen kommt. Hier wird in den Zustand «String2Lesen» umgeschaltet und so lange darin verblieben, bis wieder ein doppeltes Anführungszeichen folgt.

Der Vorteil des Automaten-Modells liegt u. a. darin, daß die Arbeitsweise in einer Tabelle dargestellt werden kann:

Zustand	geles. Zeichen	Aktion	neuer Zustand
ZeichenLesen	Hochkomma	–	String1Lesen
ZeichenLesen	Gänsefüßchen	–	String2Lesen
ZeichenLesen	andere	–	ZeichenLesen
String1Lesen	Hochkomma	neue Zeile beginnen	ZeichenLesen
String1Lesen	andere	Zeichen ausgeben	String1Lesen
String2Lesen	Gänsefüßchen	neue Zeile beginnen	ZeichenLesen
String2Lesen	andere	Zeichen ausgeben	String2Lesen

```
MODULE StringExtrakt;
  FROM InOut IMPORT OpenInput, CloseInput, Done, Read, Write, WriteString,
                    WriteLn;
  TYPE AutomatenZustand = (ZeichenLesen, String1Lesen, String2Lesen);
  VAR  Zustand : AutomatenZustand;
       Zeichen : CHAR;
  BEGIN
    WriteLn;
    WriteString("String-Extrakt aus Modula-2-Programmtexten"); WriteLn;
    WriteString("----------------------------------------"); WriteLn;
    WriteLn;
    OpenInput("MOD");
    IF NOT Done THEN WriteString("Kein File!"); WriteLn; HALT END;
    Zustand:=ZeichenLesen;
    WriteString("Die String-Konstanten:"); WriteLn;
```

```
  Read(Zeichen);
  WHILE Done DO
    CASE Zustand OF
      ZeichenLesen : IF Zeichen='"'
                     THEN Zustand:=String1Lesen
                     ELSIF Zeichen="'"
                     THEN Zustand:=String2Lesen
                     END
    | String1Lesen : IF Zeichen='"'
                     THEN WriteLn; Zustand:=ZeichenLesen
                     ELSE Write(Zeichen)
                     END
    | String2Lesen : IF Zeichen="'"
                     THEN WriteLn; Zustand:=ZeichenLesen
                     ELSE Write(Zeichen)
                     END;
    END; (* CASE *)
    Read(Zeichen)
  END; (* WHILE *)
  CloseInput
END StringExtrakt.
```

Probelauf:

String-Extrakt aus Modula-2-Programmtexten

INPUT FROM: STRINGS

Die String-Konstanten:

String-Extrakt aus Modula-2-Programmtexten

MOD
Kein File!
Die String-Konstanten:
"
'
"
'

Der Einsatz von selbstdefinierten Aufzählungstypen bringt Klarheit in Programme. In vielen anderen Programmiersprachen müßten die Zustände durch verschiedene Zahlen ausgedrückt werden.

In einem weiteren Beispiel wollen wir wieder eine Textdatei analysieren. Nur sollen diesmal die einzelnen Wörter gezählt und die mittlere Wortlänge bestimmt werden. Als Wort soll dabei jede zusammenhängende Folge von Buchstaben gelten. Am Anfang eines Wortes wird die Wortzahl erhöht. Zur Bestimmung der mittleren Wortlänge müssen zudem alle Buchstaben gezählt werden.

Unser Automat benötigt für diese Aufgabe nur zwei Zustände:

Zustand	geles. Zeichen	Aktion	neuer Zustand
ZeichenLesen	kein Buchstabe	–	ZeichenLesen
ZeichenLesen	Buchstabe	WortZahl erh. Buchstaben-zahl erh.	WortLesen
WortLesen	Buchstabe	Buchstaben-Zahl erh.	WortLesen
WortLesen	kein Buchstabe	–	ZeichenLesen

Die mittlere Wortlänge ist der Quotient aus BuchstabenZahl und WortZahl. Die Ausgabe des Ergebnisses wird über die schon bekannte Prozedur «SchreibQuotient» abgewickelt.

```
MODULE WortLaenge;
  FROM InOut IMPORT OpenInput, CloseInput, Done, Read, Write, WriteString,
                    WriteLn, WriteCard;
  TYPE AutomatenZustand = (ZeichenLesen, WortLesen);
  VAR  Zustand : AutomatenZustand;
       Zeichen : CHAR;
       WortZahl, BuchstabenZahl : CARDINAL;

  PROCEDURE SchreibQuotient(Dividend, Divisor, Feld, Stellen : CARDINAL);
    VAR i, Rest : CARDINAL;
    BEGIN
      WriteCard(Dividend DIV Divisor,Feld-Stellen-1);
      Write('.');
      Rest:=Dividend MOD Divisor;
      FOR i:=1 TO Stellen DO
        Dividend:=10*Rest;
        WriteCard(Dividend DIV Divisor,1);
        Rest:=Dividend MOD Divisor
      END (* FOR *)
    END SchreibQuotient;

  BEGIN
    WriteLn;
    WriteString("Bestimmung der mittleren Wortlänge in Texten"); WriteLn;
    WriteString("-----------------------------------------"); WriteLn;
    WriteLn;
    OpenInput("MOD");
    IF NOT Done THEN WriteString("Kein File!"); WriteLn; HALT END;
    WortZahl:=0; BuchstabenZahl:=0;
    Zustand:=ZeichenLesen;
    Read(Zeichen);
    WHILE Done DO
      CASE Zustand OF
        ZeichenLesen : IF (CAP(Zeichen)>="A") AND (CAP(Zeichen)<="Z")
                       THEN
                         INC(WortZahl);
                         INC(BuchstabenZahl);
                         Zustand:=WortLesen;
                       END
```

```
    | WortLesen     : IF (CAP(Zeichen)>="A") AND (CAP(Zeichen)<="Z")
                      THEN INC(BuchstabenZahl)
                      ELSE Zustand:=ZeichenLesen
                      END
    END; (* CASE *)
    Read(Zeichen)
  END; (* WHILE *)
  CloseInput;
  WriteString("Es wurden insgesamt "); WriteCard(WortZahl,1);
  WriteString(" Wörter gelesen."); WriteLn;
  WriteString("Zahl der Buchstaben: ");
  WriteCard(BuchstabenZahl,1);
  IF WortZahl>0
  THEN
    WriteString("Mittlere Wortlänge: ");
    SchreibQuotient(BuchstabenZahl,WortZahl,5,2);
    WriteLn
  END
END WortLaengen.
```

Testlauf:

Bestimmung der mittleren Wortlänge in Texten

INPUT FROM Wortlaengen
Es wurden insgesamt 176 Wörter gelesen.
Zahl der Buchstaben: 1106
Mittlere Wortlänge: 6.28

Aufgaben:

1. Schreiben Sie ein Programm, das einen Programmtext kopiert und dabei alle Buchstaben außerhalb von Stringkonstanten in Großbuchstaben umwandelt.
2. Aus welchem Grund ist im Programm «WortLaengen» die Anweisung «IF WortZahl > 0» notwendig?
3. Geben Sie einen Automaten an, der eine INTEGER-Zahl auf korrekte Syntax prüft. Tip: INTEGER-Zahl::=["+"|"–"]ZiffernFolge
4. Was sind die Werte von "MAX(Farbe)" und "MIN(Farbe)"?

7.2 Unterbereichstypen

Erinnern Sie sich noch an die Programmierregel, immer den knappesten Typ für eine bestimmte Aufgabe zu wählen? Bislang war nur eine Auswahl zwischen CARDINAL und INTEGER möglich. Dieses Raster ist in den meisten Anwendungsfällen aber immer noch

viel zu grob. Oft kommt es vor, daß bestimmte Größen nur innerhalb eines genau festgelegten Bereichs liegen dürfen. Genau für diesen Zweck gibt es in Modula-2 die Unterbereichstypen.

Unterbereichstyp::="[" Konstante1 ".." Konstante2 "]".

Unterbereiche können nur von aufzählbaren Typen gebildet werden. Das sind – um es nochmals zu wiederholen – CARDINAL, INTEGER, CHAR, BOOLEAN und selbstdefinierte Aufzählungstypen. Auch die Unterbereiche gehören zu den aufzählbaren Typen. Anstelle der Konstanten dürfen selbstverständlich wieder konstante Ausdrücke treten.

Beispiele:

```
TYPE Tag = [1..31];
     Monat = [1..12];
     Großbuchstaben = ["A".."Z"];
     Wochentag = (Montag, Dienstag, Mittwoch, Donnerstag, Freitag,
                  Samstag, Sonntag);  (* Selbstdef. Aufzählungstyp *)
     Arbeitstag = [Montag..Freitag];  (* Unterbereich von Wochentag *)
     Wochenende = [Samstag..Sonntag]; (* dto. *)
     WochenendeFuerStudenten = [Donnerstag..Sonntag]; (* dto *)
```

Eine ähnliche Schreibweise haben wir bereits bei der CASE-Anweisung kennengelernt. «Konstante1..Konstante2» beinhaltet alle Werte von «Konstante1» (eingeschlossen) bis «Konstante2» (eingeschlossen). «Konstante1» muß kleiner oder gleich «Konstante2» sein. Unzulässig ist demnach [«Z»..»A«]. Anstelle der Konstanten dürfen auch konstante Ausdrücke stehen, wie [2 ..10000 DIV 127].

Normalerweise wird der dem Unterbereich zugrundeliegende Typ (Basistyp) automatisch anhand der Konstanten bestimmt. Einzig bei dem Bereich, der von CARDINAL- und INTEGER-Zahlen gemeinsam abgedeckt wird (0..MAX(INTEGER)), kann eine Basistyp-Spezifikation vorgenommen werden. Soll der Unterbereich vom Typ INTEGER sein, so wird dem Unterbereich das Wort «INTEGER» vorangestellt:

TYPE KleineZahlen = INTEGER[1..10];

Für Unterbereiche sind alle Operatoren des entsprechenden Basistyps zulässig.Der Einsatz von Unterbereichstypen ist aus zwei Gründen vorteilhaft:

1. Das Festlegen zulässiger Bereiche schafft Klarheit innerhalb des Programms.
2. Die Angabe möglichst knapper Bereiche gibt dem Modula-2-System die Möglichkeit, bei der Fehlersuche in Programmen zu helfen.

Gerade der zweite Punkt ist für die Programmierpraxis von hervorragender Bedeutung. Denn das «Über-die-Grenze-Laufen» von Variablen ist eine der häufigsten Fehlerursachen in Programmen. Aus diesem Grund wollen wir ein kleines Testprogramm schreiben, das einen solchen Fehler simuliert, und die Reaktion des jeweiligen Modula-2-Systems kennenlernen.

```
MODULE UnterbereichsTest;
  TYPE ZahlBisHundert = [0..100];

VAR  KleineZahl : ZahlBisHundert;
BEGIN
  KleineZahl:=0;
  WHILE KleineZahl<200 DO INC(KleineZahl) END
END UnterbereichsTest.
```

Dieses Programm wird ohne Fehlermeldung übersetzt. Erst wenn das Programm gestartet wird, tritt beim 101. Schleifendurchlauf ein illegaler Zustand ein. Hier sollte das Programm mit einer Fehlermeldung wie «OUT OF RANGE ERROR AT 02A3B»abgebrochen werden. Da der Fehler «Bereichsüberschreitung» erst dann auftritt, wenn das Programm abläuft, spricht man von einem Laufzeitfehler (engl. runtime error). Ähnliche Fehler sind «Division durch 0» oder «Speicherüberlauf».

In jedem Modula-2-System sollte die Möglichkeit bestehen, aufgrund der zusätzlichen Angaben der Fehlermeldung (AT 02A3B) die Stelle im Programmtext zu lokalisieren, an der der Fehler auftrat. Das Werkzeug hierfür heißt «Debugger» (Entwanzer, Wanze = Fehler im Programm); seine Handhabung ist von Compiler zu Compiler verschieden. Sie sollten sich unbedingt mit diesem Hilfsprogramm vertraut machen, denn das Entwanzen umfangreicher Programme bedarf oft mehr Zeit und Energie als die gesamte Programmentwicklung.

Die Bereichsüberwachung kostet natürlich Rechenzeit. Deshalb ist es durchaus sinnvoll, bei fertig getesteten Programmen, bei denen es auf höchste Ablaufgeschwindigkeit ankommt, darauf zu verzichten. Diese Möglichkeit ist bei jedem mir bekannten Modula-2-System gegeben.

Man kann sich vorstellen, daß jeder Compiler über eine Reihe von Ein/Aus-Schaltern verfügt. Mit diesen Schaltern kann die Arbeitsweise des Compilers gezielt beeinflußt werden. Standardmäßig sind sie meist so gesetzt, daß die von Compiler und Laufzeitsystem erkennbaren Fehler angezeigt werden und zu einem Programmabbruch führen. Bei Bedarf können diese Schalter (engl. compiler switches) umgestellt werden. Auch hier unterscheidet sich die Vorgehensweise je nach Compiler.

Bei einem Teil werden die benötigten Angaben in Form speziell gekennzeichneter Kommentare (wie (*$R+ ...) direkt in den Quelltext eingegeben. Das hat einerseits den Vorteil, Teilbereiche des Programms mit unterschiedlichen Schalterstellungen versehen zu können. Der Nachteil liegt darin, daß bei langen Programmtexten oft eine Compileranweisung, wie die Schalter auch genannt werden, übersehen wird.

Eine andere Möglichkeit besteht darin, die entsprechenden Schalterstellungen beim Aufruf des Compilers anzugeben (M2 TEST.MOD /R+). Diese wirken dann für das gesamte Programm. Es ist wichtig, mit den Möglichkeiten der Compiler-Beeinflussung umgehen zu lernen. Das gilt besonders für die Fehlersuche, denn oft können noch zusätzliche Hilfen wie «Programmunterbrechung zu jedem Zeitpunkt» (letzte Rettung aus Endlosschleifen) eingeschaltet werden.

Aufgabe: Setzen Sie bei Ihrem Modula-2-Compiler die Schalter so, daß das obige Programm ohne Fehlermeldung abläuft.

7.3 Mengen

Die Mengenlehre ist eine formale Sprache, mit der sich die gesamte Mathematik darstellen läßt. Gegenüber so mächtigen Konstruktionen wie «unendliche Mengen» oder «Mengen von Mengen», nimmt sich der Teilbereich der Mengenlehre, den Modula-2 bietet, sehr bescheiden aus. Die wichtigsten Einschränkungen sind:

- ☐ Mengen können nur von aufzählbaren Typen gebildet werden.
- ☐ Die maximale Anzahl der Elemente einer Menge ist stark beschränkt (meist auf 16 oder 32).

Trotzdem lohnt sich die Beschäftigung mit Mengen in Modula-2, weil dadurch einerseits ganz elegante Programmsteuerungen mög-

lich werden, andererseits die Arbeit mit Mengen vom Computer sehr schnell erledigt werden kann.

Mengentyp::="SET" "OF" AufzählbarerTyp.

Als Typ sind nur aufzählbare Typen zugelassen, die weniger als MaxElement verschiedene Werte annehmen können. Zudem muß sichergestellt werden, daß die Ordinalwerte im Bereich von 0 bis MaxElement liegen. Als Werte eines Mengentyps kommen alle möglichen Teilmengen in Frage, auch die leere Menge, die kein Element enthält.

Beispiele:

```
TYPE KleineZahlenMenge = SET OF [0..15];
     FarbenPalette = SET OF (rot, gruen, blau, gelb, schwarz weiss);
                     oder SET OF Farbe;
     Optionen = (Fettdruck, Zeilennummern, Kommentare, Umlaute);
     Steuerung = SET OF Optionen;
```

Nicht zugelassen sind folgende Mengenkonstruktionen:

```
TYPE GrosseZahlenMenge = SET OF [0..500];
     Vokale = SET OF CHAR;
     Intervall = SET OF [0.0..1.0];
```

7.3.1 Mengenkonstante

Mengen-Konstante::=[Typname] "{" [Element] {"," Element } "}".
Element::=Konstante | Konstante1".."Konstante2.

Die Konstanten müssen vom Grundtyp der Menge sein. «Konstante1..Konstante2» ist eine Abkürzung für alle Werte von «Konstante1» bis «Konstante2». Mengenkonstanten können auch im Deklarationsteil vereinbart werden.

Beispiele:

```
CONST  Grundfarben = FarbenPalette{rot, gruen, blau};
       GeradeZahlen = KleineZahlenMenge{2,4,6,8,10,12,14};
       NormalEinstellung = Steuerung{};
```

Auch innerhalb eines Anweisungsteils wird der Mengentyp vorangestellt. Selbstverständlich kann für Variable von Mengentypen die Zuweisung verwendet werden.

Beispiele:

```
VAR Palette : FarbenPalette;
    UngeradeZahlen : KleineZahlenMenge;
    Programmsteuerung : Steuerung;

BEGIN
  Palette:=FarbenPalette(schwarz, weiss);
  UngeradeZahlen:=KleineZahlenMenge(1,3,5,7,9,11,13,15);
  Programmsteuerung:=Steuerung(Fettdruck, ZeilenNummern);
  ...
```

7.3.2 Operationen mit Mengen

Der wichtigste Operator für Mengen ist «IN», der darüber Auskunft gibt, ob ein Wert in einer Menge enthalten ist oder nicht:

Ausdruck IN Menge

Das Ergebnis dieser Operation ist vom Typ BOOLEAN. Aus diesem Grund wird der IN-Operator auch zu den relationalen Operatoren gerechnet.

Beispiele:

rot IN Palette	<–> FALSE
2*6 IN GeradeZahlen	<–> TRUE
Fettdruck IN NormalEinstellung	<–> FALSE

Mit den relationalen Operatoren =, <> oder #, <= und >= können Gleichheit, Ungleichheit und Teilmengenbeziehung geprüft werden (A und B sind Mengen):

Operator	Bedeutung
A = B	Beide Mengen enthalten dieselben Elemente.
A <> B	Beide Mengen unterscheiden sich um mindestens ein Element.
A <= B	Alle Elemente von A sind auch in B enthalten.
A >= B	Alle Elemente von B sind auch in A enthalten.

Zur Bildung von Durchschnitt, Vereinigung und Differenz werden die Operatoren *, +, – und / verwendet. Das Ergebnis der Operation ist wieder eine Menge:

Operator	Ergebnis
A * B	Alle Elemente, die sowohl A als auch B enthalten.
A + B	Alle Elemente, die in A oder B (oder beiden) sind.
A – B	Alle Elemente, die zwar in A, nicht aber in B sind.
A / B	Alle Elemente, die entweder in A oder in B sind, aber nicht in beiden Mengen.

Beispiele:

```
VAR Palette, Palette1, Palette2 : FarbenPalette;

BEGIN
  Palette1:=FarbenPalette(rot, gruen, schwarz, weiss);
  Palette2:=FarbenPalette(gruen, gelb, blau, schwarz);
  Palette:=Palette1 * Palette2;
    (* ergibt (gruen,schwarz) *)
  Palette:=Palette1 + Palette2;
    (* ergibt  (rot,gruen,gelb,blau,schwarz,weiss) *)
  Palette:=Palette1 - Palette2;
    (* ergibt  (rot,weiss) *)
  Palette:=Palette1 / Palette2;
    (* ergibt  (rot,gelb,blau,weiss) *)
  ...
```

7.3.3 Standardprozeduren für Mengen

Wie gesehen, können in Mengenkonstanten keine Variablen oder Ausdrücke auftreten. Um solche in eine Menge ein- oder auszuschließen, gibt es die Standardprozeduren «INCL» und «EXCL».

INCL(Menge,Ausdruck) schließt den Wert des Ausdrucks in Menge ein. Im Anschluß an diese Anweisung würde «Ausdruck IN Menge» den Wert «TRUE» ergeben.

EXCL(Menge,Ausdruck) entfernt – falls vorhanden – den Wert des Ausdrucks aus der Menge. Eine folgende Auswertung mit «Ausdruck IN Menge» würde das Resultat «FALSE» ergeben.

Beispiele:

```
INCL(Palette1,blau);
(* ergibt {rot,gruen,blau,schwarz,weiss} *)
EXCL(Palette1,weiss);
(* ergibt {rot,gruen,blau,schwarz} *)
```

7.3.4 Programmlister mit Optionen

Im folgenden Beispiel wird eine Menge zur Steuerung eines Programms eingesetzt. Das Programm gibt einen Modula-2-Quelltext in eine Datei aus. Dabei kann vom Anwender eine beliebige Kombination aus folgenden Optionen ausgewählt werden:

Z : Die Zeilen werden mit Nummern versehen.
S : Nach 60 Programmzeilen findet ein Seitenumbruch statt.
U : Umlaute werden umgewandelt.
A : Modula-2-Abkürzungen werden ausgeschrieben.

Die Auswahl geschieht dadurch, daß der Anwender einfach die entsprechenden Kennbuchstaben der gewünschten Optionen eingibt.

```
MODULE ListerMitOptionen;

  FROM InOut IMPORT OpenInput, OpenOutput, CloseInput, CloseOutput,
                    EOL, Done, Read, Write, WriteLn, WriteString,
                    WriteCard;
  (* EOL kann unter Umständen im Modul 'ASCII' versteckt sein *)

  CONST ZeilenProSeite = 60;

  TYPE AusgabeOptionen = (ZeilenNummern, SeitenFormatierung,
                          UmlautUmwandlung, Abkuerzungen);
       OptionenMenge = SET OF AusgabeOptionen;

  VAR  Zeilen, Seiten : CARDINAL;
       Zeichen : CHAR;
       OptionenAuswahl : OptionenMenge;

  PROCEDURE LiesOptionen(VAR Auswahl : OptionenMenge);
    VAR Eingabe : CHAR;
    BEGIN
      WriteString("Folgende Möglichkeiten stehen zur Verfügung:"); WriteLn;
      WriteLn;
      WriteString("  Z : Zeilen werden mit Nummern versehen"); WriteLn;
      WriteString("  S : Text wird seitenweise formatiert"); WriteLn;
      WriteString("  U : Umlaute werden nach ASCII umgewandelt"); WriteLn;
      WriteString("  A : Modula-2 Abkürzungen werden ausgeschrieben");
      WriteLn; WriteLn;
      WriteString("Ihre Auswahl (Z, ZS, UA ... ZSUA) : ");
      Auswahl:=OptionenMenge();
      REPEAT
        Read(Eingabe); Eingabe:=CAP(Eingabe);
        CASE Eingabe OF
          "Z" : INCL(Auswahl,ZeilenNummern)
        | "S" : INCL(Auswahl,SeitenFormatierung)
        | "U" : INCL(Auswahl,UmlautUmwandlung)
        | "A" : INCL(Auswahl,Abkuerzungen)
```

```
      | EOL :
      ELSE Write(07C)
      END (* CASE *)
    UNTIL Eingabe=EOL;
    WriteLn
  END LiesOptionen;

PROCEDURE NeueSeite(VAR SeitenZahl : CARDINAL);
  CONST FormFeed = 14C;  (* Bewirkt einen Seitenvorschub *)
  BEGIN
    IF SeitenZahl>0 THEN Write(FormFeed) END;
    INC(SeitenZahl);
    WriteString("Seite "); WriteCard(SeitenZahl,1); WriteLn;
    WriteLn
  END NeueSeite;

PROCEDURE NeueZeile(VAR ZeilenZahl : CARDINAL);
  BEGIN
    WriteCard(ZeilenZahl,4);
    Write(":")
  END NeueZeile;

BEGIN
  WriteString("Formatierte Ausgabe eines Modula-2-Quelltextes"); WriteLn;
  WriteLn;
  LiesOptionen(OptionenAuswahl);
  OpenInput("MOD");
  IF NOT Done THEN WriteString("Datei nicht vorhanden!"); HALT END;
  WriteLn;
  OpenOutput("LST");
  IF NOT Done THEN WriteString("Illegale Ausgabedatei!"); HALT END;
  Zeilen:=0; Seiten:=0;
  Zeichen:=EOL;
  WHILE Done DO
    CASE Zeichen OF
     "Ä","Ö","Ü",
     "ä","ö","ü","ß" : IF UmlautUmwandlung IN OptionenAuswahl
                       THEN
                         CASE Zeichen OF
                           "Ä" : WriteString("Ae")
                         | "Ö" : WriteString("Oe")
                         | "Ü" : WriteString("Ue")
                         | "ä" : WriteString("ae")
                         | "ö" : WriteString("oe")
                         | "ü" : WriteString("ue")
                         | "ß" : WriteString("ss")
                         END (* CASE *)
                       ELSE Write(Zeichen)
                       END
   | "&","~","#"     : IF Abkuerzungen IN OptionenAuswahl
                       THEN
                         CASE Zeichen OF
                           "&" : WriteString(" AND ")
                         | "~" : WriteString(" NOT ")
                         | "#" : WriteString("<>")
                         END (* CASE *)
                       ELSE Write(Zeichen)
                       END;
   | EOL             : IF Zeilen>0 THEN WriteLn END;
                       IF (SeitenFormatierung IN OptionenAuswahl) AND
                          (Zeilen MOD ZeilenProSeite = 0)
                       THEN NeueSeite(Seiten)
                       END;
```

```
                        INC(Zeilen);
                        IF ZeilenNummern IN OptionenAuswahl
                        THEN NeueZeile(Zeilen)
                        END
      ELSE Write(Zeichen)
      END; (* CASE *)
      Read(Zeichen)
    END; (* WHILE Done *)
    CloseInput;
    CloseOutput
END ListerMitOptionen.
```

Probelauf:

Formatierte Ausgabe eines Modula-2-Quelltextes

Folgende Möglichkeiten stehen zur Verfügung:

Z : Zeilen werden mit Nummern versehen
S : Text wird seitenweise formatiert
U : Umlaute werden nach ASCII umgewandelt
A : Modula-2-Abkürzungen werden ausgeschrieben

Auswahl (Z, ZS, UA ... ZSUA) : ZS

INPUT FROM: LISTER
OUTPUT TO: CON:

Seite 1

```
 1:MODULE ListerMitOptionen;
 2:
 3:  FROM InOut IMPORT OpenInput, OpenOutput, CloseInput, CloseOutput,
 4:                    EOL, Done, Read, Write, WriteLn, WriteString,
 5:                    WriteCard;
 6:  (* EOL kann unter Umständen im Modul 'ASCII' versteckt sein *)
 7:
 8:  CONST ZeilenProSeite = 60;
 9:
10:  TYPE AusgabeOptionen = (ZeilenNummern, SeitenFormatierung,
11:                          UmlautUmwandlung, Abkuerzungen);
12:       OptionenMenge = SET OF AusgabeOptionen;
...
```

7.3.5 Der Standardtyp BITSET

Mit «BITSET» gibt es in Modula-2 bereits eine vordefinierte Menge. Ihre Elemente sind alle CARDINAL-Zahlen bis n-1, wobei n die Wortbreite des Prozessors ist, auf dem das Modula-2-System läuft. Aus diesem Grund ist «BITSET» implementationsabhängig. Bei Konstanten vom Typ «BITSET» kann die Typangabe weggelassen werden.

Beispiel:

CONST KleinePrimZahlen = {2,3,5,7,11,13}
(* ist vom Typ BITSET *)

Selbstverständlich sind für «BITSET» alle Mengenoperatoren erlaubt. Das Interessante an «BITSET» ist jedoch, daß Werte dieses Typs üblicherweise genausoviel Speicherplatz benötigen wie Werte von CARDINAL und INTEGER (immer ein Prozessorwort). Die Elemente von Mengen werden grundsätzlich durch einzelne Bits repräsentiert. Ist ein Element in einer Menge enthalten, so ist das entsprechende Bit gesetzt (1), ansonsten nicht (0). Somit kann über «BITSET» auch auf die kleinsten Informationseinheiten zugegriffen werden. Beispielsweise können mit Hilfe des Re-Typings nun problemlos CARDINAL-Zahlen bitweise manipuliert werden.

Zu diesem Zweck schreiben wir ein eigenes Bibliotheksmodul, das alle wichtigen Operationen liefert. Hier der Definitionsteil:

```
DEFINITION MODULE BITS;
  (* Liefert Operationen, die CARDINAL-Zahlen bitweise manipulieren *)

  (* Bei älteren System muß hier die EXPORT-Liste stehen:
     EXPORT QUALIFIED MaxBits, BitPos, BitAnd, BitOr ... *)

  CONST MaxBits = 15;  (* Wortbreite-1 des Prozessors *)
  TYPE  BitPos = CARDINAL[0..MaxBits];

  PROCEDURE BitAnd(A,B : CARDINAL):CARDINAL;
    (* Funktion, liefert A AND B *)

  PROCEDURE BitOr(A,B : CARDINAL):CARDINAL;
    (* Funktion, liefert A OR B *)

  PROCEDURE BitXor(A,B : CARDINAL):CARDINAL;
    (* Funktion, liefert A XOR B *)

  PROCEDURE SetBit(n : BitPos; VAR A : CARDINAL);
    (* Prozedur, setzt das n-te Bit von A *)

  PROCEDURE ClrBit(n : BitPos; VAR A : CARDINAL);
    (* Prozedur, löscht das n-te Bit von A *)
```

```
  PROCEDURE TogBit(n : BitPos; VAR A : CARDINAL);
    (* Prozedur, schaltet das n-te Bit von A um *)

  PROCEDURE BitSet(n : BitPos; A : CARDINAL):BOOLEAN;
    (* Funktion, ergibt TRUE, wenn das n-te Bit gesetzt ist, sonst FALSE *

  PROCEDURE SHL(VAR A : CARDINAL; n : BitPos);
    (* Linksschieben um n Bits *)

  PROCEDURE SHR(VAR A : CARDINAL; n : BitPos);
    (* Rechtsschieben um n Bits *)

END BITS.
```

Solche Operationen sind dann von großer Wichtigkeit, wenn ein Computer Steueraufgaben übernehmen muß. Dann werden durch einzelne Bits bestimmte externe Aktionen ausgelöst. Alle Bit-Prozeduren, mit Ausnahme von «SHL» (Linksschieben) und «SHR» (Rechtsschieben), werden mit Mengenoperationen realisiert. Erinnern Sie sich bitte daran, daß mit

BITSET(CardinalZahl)

die «CardinalZahl» nur als «BITSET» interpretiert wird. Hier geht das Konzept des Re-Typing weit über eine Typumwandlung hinaus, denn zwischen Zahlen und Mengen von Zahlen liegen eigentlich Welten.

```
IMPLEMENTATION MODULE BITS;

  PROCEDURE BitAnd(A,B : CARDINAL) : CARDINAL;
    BEGIN
      RETURN CARDINAL(BITSET(A) * BITSET(B))
  END BitAnd;

PROCEDURE BitOr(A,B : CARDINAL) : CARDINAL;
  BEGIN
    RETURN CARDINAL(BITSET(A) + BITSET(B))
  END BitOr;

PROCEDURE BitXor(A,B : CARDINAL) : CARDINAL;
  BEGIN
    RETURN CARDINAL(BITSET(A) / BITSET(B))
  END BitXor;

PROCEDURE SetBit(n : BitPos; VAR A : CARDINAL);
  VAR T : BITSET;
  BEGIN
    T:=BITSET(A);
    INCL(T,n);
    A:=CARDINAL(T)
  END SetBit;
```

```
PROCEDURE ClrBit(n : BitPos; VAR A : CARDINAL);
  VAR T : BITSET;
  BEGIN
    T:=BITSET(A);
    EXCL(T,n);
    A:=CARDINAL(T)
  END ClrBit;

PROCEDURE TogBit(n : BitPos; VAR A : CARDINAL);
  VAR T : BITSET;
  BEGIN
    T:=BITSET(A);
    IF n IN T
    THEN EXCL(T,n)
    ELSE INCL(T,n)
    END;
    A:=CARDINAL(T)
  END TogBit;

PROCEDURE BitSet(n : BitPos; A : CARDINAL) : BOOLEAN;
  BEGIN
    RETURN n IN BITSET(A)
  END BitSet;

PROCEDURE SHL(VAR A : CARDINAL; n : BitPos);
  BEGIN
    WHILE n>0 DO ClrBit(MaxBits,A); A:=A*2; DEC(n) END
  END SHL;

PROCEDURE SHR(VAR A : CARDINAL; n : BitPos);
  BEGIN
    WHILE n>0 DO A:=A DIV 2; DEC(n) END
  END SHR;

END BITS.
```

Zur Demonstration der Bit-Operationen folgt nun ein etwas umfangreicheres Programm, das die Arbeitsweise eines Mikroprozessors simuliert. Um die internen Vorgänge besser sichtbar zu machen, werden die Zahlen im Dualsystem ausgegeben. Die Prozedur «SchreibDual» testet der Reihe nach alle Bits einer CARDINAL-Zahl durch und schreibt 1 oder 0, je nachdem, ob ein Bit gesetzt ist oder nicht.

Das Hauptprogramm läuft in einer Schleife. Mit einfachen Befehlen können auf das Ergebnis weitere Operationen angewandt werden.

```
MODULE BitTest;

  FROM InOut IMPORT ReadCard, WriteCard, WriteInt,
                    WriteLn, WriteString, Write, Read;
  FROM BITS IMPORT MaxBits, BitPos, BitAnd, BitOr, BitXor,
                   SetBit, ClrBit, TogBit, BitSet, SHL, SHR;
```

```
VAR A,B,n : CARDINAL;
    Befehl : CHAR;

PROCEDURE SchreibDual(X : CARDINAL);
  VAR i : BitPos;
  BEGIN
    FOR i:=MaxBits TO 0 BY -1 DO
      IF BitSet(i,X)
      THEN Write("1")
      ELSE Write("0")
      END (* IF *)
    END (* FOR *)
  END SchreibDual;

PROCEDURE BefehlsUebersicht;
  BEGIN
    WriteLn;
    WriteString("Das Programm simuliert die Dualarithmetik eines ");
    WriteString("Mikroprozessors."); WriteLn; WriteLn;
    WriteString("Die Befehle werden in der Form '>Befehl Argument' ");
    WriteString("eingegeben."); WriteLn; WriteLn;
    WriteString("Folgende Befehle stehen zur Verfügung:"); WriteLn;
    WriteString("  A CARDINAL -> bitweises AND"); WriteLn;
    WriteString("  O CARDINAL -> bitweises OR"); WriteLn;
    WriteString("  X CARDINAL -> bitweises XOR"); WriteLn;
    WriteString("  S n        -> setzt das n-te Bit"); WriteLn;
    WriteString("  C n        -> löscht das n-te Bit"); WriteLn;
    WriteString("  T n        -> schaltet das n-te Bit um"); WriteLn;
    WriteString("  L n        -> Linksschieben um n Bits"); WriteLn;
    WriteString("  R n        -> Rechtsschieben um n Bits"); WriteLn;
    WriteString("  B n        -> testet das n-te Bit"); WriteLn;
    WriteString("  ?          -> bringt diese Übersicht"); WriteLn;
    WriteString("  Q          -> beendet das Programm"); WriteLn;
    WriteLn
  END BefehlsUebersicht;

BEGIN
  WriteString("Demonstration der Bit-Operationen"); WriteLn;
  WriteString("---------------------------------"); WriteLn;
  BefehlsUebersicht;
  A:=0;
    REPEAT
      WriteString("= "); SchreibDual(A);
      WriteString(" CARD: "); WriteCard(A,6);
      WriteString(" INT: "); WriteInt(INTEGER(A),7);
      WriteString("  >");
      Read(Befehl); Befehl:=CAP(Befehl); ReadCard(B);
      CASE Befehl OF
        "A","O","X" : CASE Befehl OF
                        "A" : A:=BitAnd(A,B)
                      | "O" : A:=BitOr(A,B)
                      | "X" : A:=BitXor(A,B)
                      END; (* CASE *)
                      Write(Befehl); Write(" ");
                      SchreibDual(B)
      | "S","C","T",
        "L","R"       : IF B<=MaxBits
                        THEN
                          n:=B;
                          CASE Befehl OF
```

```
                        "S" : SetBit(n,A)
                      | "C" : ClrBit(n,A)
                      | "T" : TogBit(n,A)
                      | "L" : SHL(A,n)
                      | "R" : SHR(A,n)
                    END; (* CASE *)
                  ELSE WriteString("Argument zu groß!")
                  END
    | "B"         : IF B<=MaxBits
                  THEN
                    n:=B;
                    IF BitSet(n,A)
                    THEN WriteString("  TRUE")
                    ELSE WriteString("  FALSE")
                    END
                  ELSE WriteString("Argument zu groß!")
                  END
    | "?"         : BefehlsUebersicht
    | "Q"         :
    ELSE WriteString("???")
    END; (* CASE *)
    WriteLn
  UNTIL Befehl="Q"
END BitTest.
```

Testlauf:

```
Demonstration der Bit-Operationen
---------------------------------

Das Programm simuliert die Dualarithmetik eines Mikroprozessors.

Die Befehle werden in der Form '>Befehl Argument' eingegeben.

Folgende Befehle stehen zur Verfügung:
  A CARDINAL -> bitweises AND
  O CARDINAL -> bitweises OR
  X CARDINAL -> bitweises XOR
  S n        -> setzt das n-te Bit
  C n        -> löscht das n-te Bit
  T n        -> schaltet das n-te Bit um
  L n        -> Linksschieben um n Bits
  R n        -> Rechtsschieben um n Bits
  B n        -> testet das n-te Bit
  ?          -> bringt diese Übersicht
  Q          -> beendet das Programm

= 0000000000000000 CARD:      0 INT:       0  >S 15
= 1000000000000000 CARD:  32768 INT:  -32768  >O 149
O 0000000010010101
= 1000000010010101 CARD:  32917 INT:  -32619  >R 4
= 0000100000001001 CARD:   2057 INT:    2057  >X 255
X 0000000011111111
= 0000100011110110 CARD:   2294 INT:    2294  >A 255
A 0000000011111111
= 0000000011110110 CARD:    246 INT:     246  >T 14
= 0100000011110110 CARD:  16630 INT:   16630  >T 15
= 1100000011110110 CARD:  49398 INT:  -16138  >B 7  TRUE
= 1100000011110110 CARD:  49398 INT:  -16138  >Q
```

7.4 Felder

Wird eine bestimmte Anzahl von Variablen eines Typs benötigt, so benutzt man hierfür ein Datenfeld (engl. ARRAY).

Feldtyp::="ARRAY" Indextyp {"," Indextyp} "OF" Typ.
Indextyp::=Unterbereichs-Typ | Aufzählungstyp | CHAR | BOOLEAN.

Beispiele:

```
TYPE MessReihe = ARRAY[1..100] OF REAL;
     TextZeile = ARRAY[0..80] OF CHAR;
     TextSeite = ARRAY[1..60] OF TextZeile;
     PixelBildschirm = ARRAY[1..1000] OF BITSET;
     BuchstabenHaeufigkeit = ARRAY CHAR OF CARDINAL;
     Matrix = ARRAY[1..3],[1..3] OF REAL;
```

Die Anzahl der Indextypen gibt die Dimension des Feldes an. In den Beispielen sind alle Felder eindimensional, Ausnahme ist die zweidimensionale «Matrix».Die einzige Operation für Variable eines Feldtyps ist die Zuweisung.

VAR A, B : Matrix;
...
B:=A; ...

Der Zugriff auf die einzelnen Komponenten eines Feldes (Feldelemente) geschieht durch die Angabe der aktuellen Indizes in eckigen Klammern: Feldname[Ausdruck,Ausdruck...]. Die Werte der einzelnen Ausdrücke müssen kompatibel zu den angegebenen Indextypen sein. Auf die Feldelemente dürfen alle Operationen angewandt werden, die für Daten des Feldtyps zulässig sind.

Beispiele:

```
VAR Messung : MessReihe;
    A, B : Matrix;
    Zaehlung : BuchstabenHaeufigkeit;
...
INC(Zaehlung['A']);
WriteReal(Messung[35],10);
A[1,3]:=14+B[2,2]/20;
```

Als erstes Beispiel für Felder wollen wir ein Programm schreiben, das die Buchstabenhäufigkeiten in einem Text bestimmt. Ohne Felder wäre dieses Unterfangen sehr aufwendig:

VAR AnzahlA, AnzahlB, AnzahlC, AnzahlD ... : CARDINAL;

Da alle Daten von gleichen Typ CARDINAL sind, bietet sich folgende Felddefinition an:

VAR Anzahl : ARRAY ['A'..'Z'] OF CARDINAL;

Das Programm selbst realisieren wir in zwei Prozeduren, wovon die eine mit dem Namen «LiesText» das Lesen des Textes und Zählen der Buchstaben übernimmt, während die andere («Ausgabe») eine ansprechende Bildschirmausgabe besorgt.

```
MODULE BuchstabenHaeufigkeit;

  FROM InOut IMPORT OpenInput, CloseInput, Read, Write, WriteLn,
                    WriteString, WriteCard, Done;

  VAR Anzahl : ARRAY ['A'..'Z'] OF CARDINAL;

  PROCEDURE LiesText;
    VAR Zeichen : CHAR;
    BEGIN
      FOR Zeichen:='A' TO 'Z' DO Anzahl[Zeichen]:=0 END; (* Feld löschen *)
      OpenInput("");
      IF NOT Done THEN WriteString("Keine Datei!"); WriteLn; HALT END;
      Read(Zeichen);
      WHILE Done DO
        Zeichen:=CAP(Zeichen);     (* In Großbuchstaben umwandeln *)
        IF (Zeichen>='A') AND (Zeichen<='Z')
        THEN INC(Anzahl[Zeichen])
        END; (* IF *)
        Read(Zeichen)
      END; (* WHILE *)
      CloseInput
    END LiesText;

  PROCEDURE Ausgabe;
    VAR i, j , Spalten, Zeilen, FeldelementZahl : CARDINAL;
        Zeichen : CHAR;
    BEGIN
      FeldelementZahl:=1+ORD('Z')-ORD('A');
      Spalten:=2; Zeilen:=(FeldelementZahl+Spalten-1) DIV Spalten;
      FOR i:=1 TO Zeilen DO
        FOR j:=1 TO Spalten DO
          Zeichen:=CHR(ORD('A')+(j-1)*Zeilen+i-1);
          IF Zeichen<='Z'
          THEN
            Write(Zeichen); WriteString(" -> ");
            WriteCard(Anzahl[Zeichen],10);
```

```
          WriteString("          ")
        END (* IF *)
      END; (* FOR j *)
      WriteLn
    END (* FOR i *)
  END Ausgabe;

BEGIN  (* BuchstabenHaeufigkeit *)
  WriteString("Buchstabenhäufigkeit in einem Text"); WriteLn;
  WriteString("----------------------------------"); WriteLn;
  WriteLn;
  LiesText;
  Ausgabe
END BuchstabenHaeufigkeit.
```

Probelauf:

Buchstabenhäufigkeit in einem Text

INPUT FROM BUCHST.MOD

A –>	54	N –>	85
B –>	14	O –>	31
C –>	34	P –>	15
D –>	38	Q –>	0
E –>	131	R –>	53
F –>	19	S –>	24
G –>	16	T –>	60
H –>	40	U –>	19
I –>	79	V –>	4
J –>	4	W –>	19
K –>	4	X –>	4
L –>	42	Y –>	1
M –>	8	Z –>	30

Die Prozedur «Ausgabe» ist universell einsetzbar, wenn ein eindimensionales Feld mehrspaltig ausgegeben werden soll.

7.4.1 Offene Felder als Parameter

Dem häufigen Wunsch nach möglichst universell einsetzbaren Prozeduren steht der Zwang entgegen, daß allen Parametern eine Typangabe folgen muß. Verwendet man als Parameter ARRAY-Typen, so ist die Prozedur auf die bei der Typdefinition festgelegten Feldgrenzen beschränkt. Um diese Einengung umgehen zu können, bietet Modula-2 die sogenannten «offenen Feld-Parameter». Hier wird als Typ die Konstruktion «ARRAY OF ...» verwendet.

Beispiel:

PROCEDURE Ausgabe(Zahlenfeld : ARRAY OF CARDINAL);

Innerhalb der Prozedur ist das übergebene Feld folgendermaßen zu interpretieren:

ARRAY[0..x] OF CARDINAL;

Je nach aktuellem Parameter ändert sich die Größe x. Ihren Wert gibt die Standardprozedur «HIGH» an, deren einziger Parameter der Name des offenen Feldes ist.

```
VAR  Feld1 : ARRAY[1..10] OF CARDINAL;
     Feld2 : ARRAY['A'..'Z'] OF CARDINAL;
     Feld3 : ARRAY[-10..10] OF CARDINAL;
```

Innerhalb der Prozedur «Ausgabe» gelten nun – je nachdem, welche Variable zur Bearbeitung übergeben wird – folgende Größen:

Zahlenfeld	HIGH(Zahlenfeld)	Zahlenfeld[0]
Feld1	9	Feld1[1]
Feld2	25	Feld2['A']
Feld3	20	Feld3[-10]

Soll beispielsweise das gesamte Zahlenfeld ausgegeben werden, so könnte die Prozedur «Ausgabe» folgende Gestalt haben:

```
PROCEDURE Ausgabe(Zahlenfeld : ARRAY OF CARDINAL);
  VAR i : CARDINAL;
  BEGIN
    FOR i:=0 TO HIGH(Zahlenfeld) DO
      WriteCard(Zahlenfeld[i],10);
      WriteLn
    END (* FOR *)
  END Ausgabe;
```

Hinweis: Offene Feldparameter sind nur bei eindimensionalen Feldern zulässig.

Aufgabe: Bestimmen Sie HIGH(Zahlenfeld) für folgende Parameter:

```
TYPE Farben = (rot,gruen,blau,gelb,schwarz,weiss);

VAR  Feld4 : ARRAY Farben OF CARDINAL;
     Feld5 : ARRAY BOOLEAN OF CARDINAL;
     Feld6 : ARRAY[20..30] OF CARDINAL;
```

7.4.2 Zeichenketten in Modula-2

Eine ganz besondere Rolle spielen in Modula-2 Felder der Form

ARRAY[0..MaxZeichen] OF CHAR

Es handelt sich hierbei um Zeichenketten, sogenannte «Strings», in denen eine bestimmte Anzahl (höchstens MaxZeichen+1) Zeichen gespeichert werden können. Da in den seltensten Fällen alle Zeichen des Feldes belegt werden, wird das Ende einer Zeichenkette dadurch angezeigt, daß in dem Feldelement nach dem letzten gültigen Zeichen das Steuerzeichen 0C abgelegt wird. Aus diesem Grund kann die Zuweisung an eine Variable dieses Typs wesentlich vereinfacht werden.

```
CONST MaxZeichen = 80;
VAR Zeile : ARRAY[0..MaxZeichen] OF CHAR;
...

statt Zeile[0]:='M'; Zeile[1]:='o'; ... Zeile[6]:=0C
kann die Zuweisung einer Stringkonstanten an das komplette Feld
erfolgen:

  Zeile:="Modula"
```

Bei der Ausführung dieser Zuweisung werden die restlichen Feldelemente automatisch mit 0C aufgefüllt.

Das Konzept der offenen Feldparameter erlaubt nun die Bereitstellung von Prozeduren zum Verarbeiten von Zeichenketten, ohne auf die jeweiligen Dimensionierungen Rücksicht nehmen zu müssen. Da auch Stringkonstante intern als «ARRAY ... OF CHAR» dargestellt werden, können sie als offene Feldparameter übergeben werden.

Beispiele:

```
PROCEDURE StringLaenge(Zeile : ARRAY OF CHAR) : CARDINAL;
  VAR i : CARDINAL;
  BEGIN
    i:=0;
    WHILE (i<=HIGH(Zeile)) AND (Zeile[i]<>0C) DO INC(i) END;
    RETURN i
  END StringLaenge;
```

Die Laufvariable i zeigt nach Beendigung auf das Zeichen nach dem letzten gültigen. Da bei 0 zu zählen begonnen wird, entspricht i der Länge des Strings.

```
PROCEDURE Concat(S1,S2 : ARRAY OF CHAR; VAR S : ARRAY OF CHAR);
  (* Setzt den String S aus den beiden Teilstrings S1 und S2
     zusammen *)
  VAR LaengeS1, LaengeS2, LengeS, i : CARDINAL;
  BEGIN
    LaengeS1:=StringLaenge(S1); LaengeS2:=StringLaenge(S2);
    IF LaengeS1>HIGH(S)+1      (* Falls S1 nicht ganz in S paßt *)
    THEN LaengeS1:=HIGH(S)+1  (* wird S1 entsprechend verkürzt *)
    END;
    FOR i:=1 TO LaengeS1 DO S[i-1]:=S1[i-1] END; (* S1 wird kopiert *)
    IF LaengeS2>HIGH(S)+1-LaengeS1       (* Paßt S2 noch dazu? *)
    THEN LaengeS2:=HIGH(S)+1-LaengeS1    (* Wenn nein, dann verkürzen *)
    END; (* IF *)  (* und kopieren *)
    FOR i:=1 TO LaengeS2 DO S[LaengeS1+i-1]:=S2[i-1] END;
    IF LaengeS1+LaengeS2<=HIGH(S)        (* Ist S völlig gefüllt? *)
    THEN S[LaengeS1+LaengeS2]:=0C        (* Ansonsten 0C anhängen *)
    END (* IF *)
  END Concat;
```

Es muß sichergestellt werden, daß alle Zuweisungen innerhalb der erlaubten Grenzen liegen. Aus diesem Grund macht die Prozedur einen eher umständlichen Eindruck.

```
PROCEDURE Pos(Suchstring, String : ARRAY OF CHAR) : CARDINAL;
  (* Gibt das erste Auftreten des Suchstrings in einem String
     zurück. 0 bedeutet, daß der Suchstring nicht enthalten ist *)
  VAR LaengeSuchstring, LaengeString, i, j : CARDINAL;
  BEGIN
    LaengeSuchstring:=StringLaenge(Suchstring);
    LaengeString:=StringLaenge(String);
    i:=0;  (* i geht die einzelnen Zeichen von String durch *)
    LOOP
      (* Falls der Suchstring schon von der Länge her nicht mehr
         passen kann, war die Suche erfolglos *)
      IF i+LaengeSuchstring>LaengeString THEN RETURN 0 END;
      j:=0;  (* j geht die einzelnen Zeichen von SuchString durch *)
      (* j wird so lange erhöht, wie die Zeichen übereinstimmen *)
      WHILE (j<LaengeSuchstring) AND (String[i+j]=Suchstring[j]) AND DO
        INC(j)
      END; (* WHILE *)
      (* Falls alle Zeichen gepaßt haben, ist das Ergebnis i+1 *)
      IF j=LaengeSuchstring THEN RETURN i+1 END;
      INC(i)  (* Ansonsten ein neuer Versuch mit i+1 *)
    END (* LOOP *)
  END Pos;
```

Aufgaben:

1. Schreiben Sie ein Programm, das die obigen Prozeduren testet.
2. Schreiben Sie eine Prozedur «Print», die einen String in einem Feld von n Zeichen rechtsbündig ausdruckt.
3. Schreiben Sie ein Bibliotheksmodul «Strings», das die obigen Prozeduren exportiert.

7.4.3 Zahlenkonvertierung

Eine interessante Anwendung ist die Eingabe von Zahlen, wenn verschiedene Zahlenbasen zugelassen sind. Wir gehen dabei so vor, daß wir die Zahl zunächst in eine Zeichenkette einlesen und dann in eine entsprechende CARDINAL-Zahl umwandeln. Folgende Zahlenarten wollen wir zulassen:

Zahl::=Dezimalzahl | Oktalzahl | Binärzahl | Hexzahl.
Binärzahl::=Binärziffer{Binärziffer}"B".
Binärziffer::="0" | "1".
Oktalzahl::=Oktalziffer{Oktalziffer}"O".
Oktalziffer::=Binärziffer|"2"|"3"|"4"|"5"|"6"|"7".
Dezimalzahl::=Ziffer{Ziffer}["D"].
Ziffer::=Oktalziffer|"8"|"9".
Hexzahl::=Hexziffer{Hexziffer}"H".
Hexziffer::=Ziffer|"A"|"B"|"C"|"D"|"E"|"F".

Hier folgt nun zunächst die Prozedur «StringToCard» zum Konvertieren einer Zahl als Zeichenfolge in ihr CARDINAL-Äquivalent. Da die Basis erst am Ende der Zahl angegeben wird, wollen wir durch eine CARDINAL-Variable «Fehler» den Erfolg der Umwandlung mitteilen.

Fehler:
- 0 -> alles in Ordnung
- 1 -> illegale Zahlenbasis
- 2 -> illegale Zeichen in der Zahl
- 3 -> Ziffer paßt nicht zur Basis
- 4 -> CARDINAL-Überlauf
- 5 -> Leerstring wurde übergeben

```
PROCEDURE StringToCard(String : ARRAY OF CHAR; VAR Zahl, Fehler : CARDINAL);
  VAR Basis, Ziffer, i, j : CARDINAL;
  BEGIN
    i:=0; (* Zunächst wird die Basis bestimmt *)
    WHILE (i<=HIGH(String)) AND (String[i]<>0C) DO INC(i) END;
    IF i=0 (* Es wurde ein leerer String übergeben *)
    THEN Fehler:=5; RETURN
    END; (* IF *)
    CASE String[i-1] OF
      'D','d'  : Basis:=10
    | '0'..'9' : Basis:=10; INC(i)
    | 'B','b'  : Basis:=2
    | 'O','o'  : Basis:=8
    | 'H','h'  : Basis:=16
    ELSE Fehler:=1; RETURN
```

```
    END; (* CASE *)
    Zahl:=0; (* Jetzt wird die Zahl von vorne aufgebaut *)
    FOR j:=0 TO i-2 DO
      CASE String[j] OF
        '0'..'9' : Ziffer:=ORD(String[j])-ORD('0')
      | 'A'..'F','a'..'f' : Ziffer:=ORD(CAP(String[j]))-ORD('A')+10
      ELSE Fehler:=2; RETURN
      END; (* CASE *)
      IF Ziffer>=Basis THEN Fehler:=3; RETURN
      ELSIF Zahl>(MAX(CARDINAL)-Ziffer) DIV Basis
      THEN Fehler:=4; RETURN
      ELSE Zahl:=Basis*Zahl+Ziffer
      END (* IF *)
    END; (* FOR *)
    Fehler:=0
  END StringToCard;
```

Ein kleines Testprogramm kann man sich ganz einfach erstellen:

```
MODULE Konvertierung;

  FROM InOut IMPORT ReadString, WriteCard, WriteLn, WriteString;

  PROCEDURE StringToCard ... (wie oben)

  VAR String : ARRAY[0..20] OF CHAR;
      Card : CARDINAL;
      Fehler : CARDINAL;

  BEGIN
    WriteString("Bitte eine CARDINAL-Zahl aus dem"); WriteLn
    WriteString("B)inär-, O)ktal-, D)ezimal- oder H)exadezimalsystem.");
    WriteLn; WriteLn;
    WriteString("Fügen Sie den Kennbuchstaben der Zahlenbasis an die");
    WriteLn;
    WriteString("Ziffernfolge an (Beispiel: FF08H)."); WriteLn;
    WriteLn;
    WriteString("Ihre Zahl: ");
    ReadString(String); WriteLn;
    StringToCard(String,Card,Fehler);
    CASE Fehler OF
      0 : WriteString("Ihre Zahl im Dezimalsystem: ");
          WriteCard(Card,1)
    | 1 : WriteString("Illegale Zahlenbasis")
    | 2 : WriteString("Illegale Zeichen in der Zahl")
    | 3 : WriteString("Ziffer paßt nicht zur Basis")
    | 4 : WriteString("CARDINAL-Überlauf")
    | 5 : WriteString("Leerstring wurde übergeben")
    END; (* CASE *)
    WriteLn
  END Konvertierung.
```

Probelauf:

```
Bitte eine CARDINAL-Zahl aus dem
B)inär-, O)ktal-, D)ezimal- oder H)exadezimalsystem.
```

Fügen Sie den Kennbuchstaben der Zahlenbasis an die Ziffernfolge an (Beispiel: FF08H).

Ihre Zahl: 1020001B
Ziffer paßt nicht zur Basis

Aufgaben:

1. Schreiben Sie eine Prozedur «CardToString(VAR S : ARRAY OF CHAR; Zahl, Basis : CARDINAL)».
2. Schreiben Sie eine Prozedur «LiesCard», die keine Fehleingaben zuläßt (und auf «StringToCard» zurückgreift), und bauen Sie diese Prozedur in das Programm «BitTest» ein.

7.4.4 Datensuche in Feldern

Will man ein Feld daraufhin untersuchen, ob ein bestimmtes Element enthalten ist oder nicht, so kann man den Suchaufwand ganz wesentlich reduzieren, wenn die Feldelemente in geordneter Reihenfolge vorliegen. Ist ein Feld sortiert, dann kann derselbe Algorithmus verwendet werden, den man bei der Suche nach einem Namen im Telefonbuch anwendet. Hier schlägt man erst einmal in der Mitte auf. Aufgrund der alphabetischen Anordnung kann man nun sofort sagen, ob sich der gesuchte Name in der ersten oder zweiten Hälfte befindet. Dieses Verfahren wird so lange wiederholt, bis der gewünschte Teilnehmer gefunden wird.

Dieser Algorithmus, bei dem die Anzahl der in Frage kommenden Elemente bei jedem Schritt halbiert wird, hat den Namen «binäre Suche» erhalten.

Angenommen, wir wollen prüfen, ob sich die CARDINAL-Zahl «Gesucht» im Zahlenfeld «Zahlen» befindet. Dazu übertragen wir den obigen Algorithmus in eine boolesche Funktionsprozedur «vorhanden».

```
PROCEDURE vorhanden(Zahlen : ARRAY OF CARDINAL; Gesucht : CARDINAL) :
 BOOLEAN;
VAR Oben, Unten, Mitte : CARDINAL;
BEGIN
  Unten:=0; Oben:=HIGH(Zahlen); (* Am Anfang ist es das ganze Feld *)
  REPEAT
    Mitte:=(Unten+Oben) DIV 2; (* Die Mitte bestimmen *)
    IF Gesucht<Zahlen[Mitte]   (* Gesucht ist in der linken Hälfte *)
    THEN Oben:=Mitte-1         (* also Oben entsprechend setzen *)
    ELSIF Gesuch>Zahlen[Mitte] (* oder in der rechten Hälfte? *)
```

```
    THEN Unten:=Mitte+1          (* dann wird Unten korrigiert *)
    ELSE RETURN TRUE             (* ansonsten ist es die Zahl! *)
  UNTIL Oben<Unten;
  RETURN FALSE                   (* Pech gehabt *)
END vorhanden;
```

Das funktioniert allerdings nur, wenn das Feld «Zahlen» auch aufsteigend sortiert ist. Das Sortieren von Feldern ist eine eigene Wissenschaft. In diesem Buch werden Sie zwei grundverschiedene Algorithmen kennenlernen. Der eine ist leichtverständlich, dafür aber langsam und nur für kleine Felder zu gebrauchen. Der andere gehört zum Schnellsten, was die Informatik derzeit zu bieten hat. Er befindet sich im Kapitel über Rekursion und verlangt schon etwas mehr Mühe, um begriffen zu werden.

Der einfache Algorithmus arbeitet so, wie ein Kartenspieler sein Blatt sortiert, nachdem er alle Karten auf einmal aufgenommen hat. Dazu bringt er zuerst die höchste Karte an die erste Stelle, dann die zweithöchste an die zweite usw., bis alle Karten geordnet sind. Wir wollen das Verfahren wieder an einem CARDINAL-Feld durchspielen:

```
PROCEDURE Sortiere(VAR Zahlen : ARRAY OF CARDINAL);
  VAR i, j : CARDINAL; (* zum Durchsuchen des Feldes *)
      kleinstes : CARDINAL; (* zeigt auf das kleinste Feldelement *)
      temp : CARDINAL; (* zum Vertauschen *)
  BEGIN
    FOR i:=0 TO HIGH(Zahlen)-1 DO
  (* Alle Zahlen bis auf die letzte werden betrachtet. Wenn alle anderen
     Zahlen an der richtigen Stelle sind,  ist es die letzte automatisch *)
      kleinstes:=i; (* Es könnte ja sein, daß Zahlen[i] bereits stimmt *)
      FOR j:=i+1 TO HIGH(Zahlen) DO (* j klappert den Rest ab *)
        IF Zahlen[j]<Zahlen[kleinstes]  (* Es gibt noch ein kleineres *)
        THEN kleinstes:=j
        END; (* IF *)
      END; (* FOR j *)
      IF i<>kleinstes (* Muß etwas vertauscht werden? *)
      THEN temp:=Zahlen[i]; Zahlen[i]:=Zahlen[j]; Zahlen[j]:=temp
      END (* IF *)
    END (* FOR i *)
  END Sortiere;
```

Im folgenden Beispielprogramm sind beide Prozeduren enthalten. Da das Feld nicht in jedem Fall vollständig gefüllt sein muß, wird jeweils die aktuelle Feldgröße in «A» übergeben. Der Anweisungsteil des Moduls besteht nur aus Prozeduraufrufen – ein typisches Beispiel für umfangreichere Programme.

```
MODULE ZahlenFelder;
  FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;

  CONST MaxZahlen = 100;
  VAR   Feld : ARRAY[1..MaxZahlen] OF CARDINAL;
        Anzahl : CARDINAL;

  PROCEDURE Eingabe(VAR F : ARRAY OF CARDINAL; VAR A : CARDINAL);
    VAR NaechsteZahl : CARDINAL;
    BEGIN
      A:=0;
      WriteString("Bitte geben Sie eine Folge von max. ");
      WriteCard(MaxZahlen,1);
      WriteString(" CARDINAL-Zahlen ein."); WriteLn;
      WriteString("Ende der Zahlenfolge = 0"); WriteLn;
      ReadCard(NaechsteZahl);
      WHILE (A<=HIGH(F)) AND (NaechsteZahl>0) DO
        F[A]:=NaechsteZahl;
        ReadCard(NaechsteZahl);
        INC(A)
      END; (* WHILE *)
      WriteLn
    END Eingabe;

  PROCEDURE Ausgabe(F : ARRAY OF CARDINAL; A : CARDINAL);
    CONST MaxSpalten = 10; (* 80-Zeichen-Bildschirm *)
          BildschirmZeilen = 24;
    VAR i, j : CARDINAL;
        ZeilenZahl : CARDINAL;
        SpaltenZahl : [1..MaxSpalten];
    BEGIN
      IF (A+BildschirmZeilen-1) DIV BildschirmZeilen>MaxSpalten
      THEN SpaltenZahl:=MaxSpalten
      ELSE SpaltenZahl:=(A+BildschirmZeilen-1) DIV BildschirmZeilen
      END; (* IF *)
      ZeilenZahl:=(A+SpaltenZahl-1) DIV SpaltenZahl;
      FOR i:=1 TO ZeilenZahl DO
        FOR j:=0 TO SpaltenZahl-1 DO
          IF i+j*ZeilenZahl<=A THEN WriteCard(F[i+j*ZeilenZahl-1],8) END
        END; (* FOR j *)
        IF SpaltenZahl<MaxSpalten THEN WriteLn END
      END; (* FOR i *)
      WriteLn
    END Ausgabe;

  PROCEDURE Sortierung(VAR F : ARRAY OF CARDINAL; A : CARDINAL);
    VAR i, j, kleinstes : CARDINAL;
        temp : CARDINAL;

    BEGIN (* Sortierung *)
      FOR i:=0 TO A-2 DO
        kleinstes:=i;
        FOR j:=i+1 TO A-1 DO
          IF F[j]<F[kleinstes] THEN kleinstes:=j END
        END; (* FOR j *)
        IF i<>kleinstes
        THEN temp:=F[i]; F[i]:=F[kleinstes]; F[kleinstes]:=temp
        END
      END (* FOR i *)
    END Sortierung;
```

```
PROCEDURE TesteZahlen(F : ARRAY OF CARDINAL; A : CARDINAL);
  VAR TestZahl : CARDINAL;

  PROCEDURE vorhanden(F : ARRAY OF CARDINAL; A : CARDINAL;
                      Gesucht : CARDINAL) : BOOLEAN;
    VAR Oben, Unten, Mitte : CARDINAL;
    BEGIN
      Unten:=0; Oben:=A-1;
      REPEAT
        Mitte:=(Unten+Oben) DIV 2;
        IF Gesucht<F[Mitte] THEN Oben:=Mitte-1
        ELSIF Gesucht>F[Mitte] THEN Unten:=Mitte+1
        ELSE RETURN TRUE
        END (* IF *)
      UNTIL Oben<Unten;
      RETURN FALSE
    END vorhanden;

  BEGIN (* TesteZahlen *)
    WriteLn;
    WriteString("Suche von Zahlen im geordneten Feld"); WriteLn;
    WriteString("Geben Sie beliebige natürliche Zahlen ein (0=Ende)");
    WriteLn;
    REPEAT
      WriteString("x = "); ReadCard(TestZahl);
      WriteString(" ist ");
      IF NOT vorhanden(F,A,TestZahl) THEN WriteString("nicht ") END;
      WriteString("vorhanden."); WriteLn
    UNTIL TestZahl=0
  END TesteZahlen;

BEGIN (* ZahlenFelder *)
  Eingabe(Feld,Anzahl);
  Sortierung(Feld,Anzahl);
  Ausgabe(Feld,Anzahl);
  TesteZahlen(Feld,Anzahl)
END ZahlenFelder.
```

Probelauf:

```
Bitte geben Sie eine Folge von max. 100 CARDINAL-Zahlen ein.
Ende der Zahlenfolge = 0
1
23
3
4
15
22
8
0

      1
      3
      4
```

```
 8
15
22
23
```

```
Suche von Zahlen im geordneten Feld
Geben Sie beliebige natürliche Zahlen ein (0=Ende)
x = 19 ist nicht vorhanden
x = 22 ist vorhanden
...
```

Aufgaben:

1. Prüfen Sie, inwieweit die folgende Alternative zur Funktionsprozedur «vorhanden» dieselben Resultate liefert. Weshalb ist sie der ursprünglichen Version vorzuziehen?

```
PROCEDURE vorhanden(F : ARRAY OF CARDINAL; A : CARDINAL;
                    Gesucht : CARDINAL) : BOOLEAN;
  VAR Oben, Unten, Mitte : CARDINAL;

BEGIN
  Unten:=0; Oben:=A-1;
  WHILE Unten<>Oben DO
    Mitte:=(Unten+Oben) DIV 2;
    IF Gesucht>F[Mitte]
    THEN Unten:=Mitte+1
    ELSE Oben:=Mitte
    END (* IF *)
  END; (* WHILE *)
  RETURN F[Unten]=Gesucht
END vorhanden;
```

2. Wie kann diese Version realisiert werden, wenn nur eine Funktion «kleiner» zum Vergleich zweier Feldelemente vorhanden ist, nicht jedoch ein Test auf Gleichheit?

7.4.5 Ein Modula-2-Ausdruckprogramm

Eine wichtige Hilfe bei der Programmierarbeit ist ein Programm zum Ausdrucken eines Quelltextes. Eine sinnvolle Einrichtung dabei ist die Numerierung der einzelnen Programmzeilen. Das Hervorheben der Schlüsselwörter erleichtert den Überblick – aber nur, wenn sie in Kommentaren und Zeichenketten unterdrückt werden. Wie bereits im Kapitel über Module gezeigt, muß bei der

Analyse von Kommentaren ein Zeichen über das gerade aktuelle hinausgelesen werden.

Die Arbeit der Textanalyse verrichtet ein endlicher Automat mit den Zuständen «ZeichenLesen», «WortLesen», «String1», «String2» und «Kommentar». Eine Buchstabenfolge wird in der Stringvariablen «Puffer» gespeichert. «ZeichenZahl» gibt die augenblickliche Stringlänge an.

Alter Zustand	Zeichen	Aktion	Neuer Zustand
ZeichenLesen	Buchstabe	ZeichenZahl:=1 Puffer[ZeichenZahl]:=Zeichen	WortLesen
	'"'	-	String1
	"'"	-	String2
	'('	Falls Naechstes='*' dann KommentarTiefe:=1	Kommentar
		ansonsten	ZeichenLesen
	sonst		ZeichenLesen
WortLesen	Buchstabe	INC(ZeichenZahl) Puffer[ZeichenZahl]:=Zeichen	WortLesen
	sonst	Falls Puffer Schlüsselwort dann hervorheben ansonsten ausgeben	ZeichenLesen
String1	'"'		ZeichenLesen
	sonst		String1
String2	"'"		ZeichenLesen
	sonst		String2
Kommentar	'('	Falls Naechstes='*' dann INC(KommentarTiefe)	Kommentar
	'*'	Falls Naechstes=')' dann DEC(KommentarTiefe) Bei KommentarTiefe=0	ZeichenLesen
		ansonsten	Kommentar

Eigentlich müßte das Zeichen, das das Ende eines Bezeichners anzeigt, wieder zurückgeschrieben werden, da es unter Umständen den Beginn eines Kommentars oder einer Zeichenkette anzeigen könnte. Wenn wir jedoch davon ausgehen, daß ausschließlich hauptsächlich korrekte Programmtexte bearbeitet werden und zwischen Bezeichner und Kommentar mindestens ein Leerzeichen steht, kann dieser Mangel in Kauf genommen werden. Andernfalls muß die Ausgabe etwas aufwendiger gestaltet und für das zurückgeschriebene Zeichen unterdrückt werden.

Die Frage, ob es sich bei dem isolierten Bezeichner um ein Schlüsselwort handelt, beantwortet die boolesche Funktionsproze-

dur «ReserviertesWort». Diese wiederum greift auf ein Feld zu, in dem alle Schlüsselwörter alphabetisch sortiert abgelegt sind. Da dieses Feld erst entsprechend initialisiert werden muß, bietet sich der Einsatz eines weiteren lokalen Moduls an, das ausschließlich die Funktionsprozedur «ReserviertesWort» exportiert.

Die (zu «ReserviertesWort») lokale Funktionsprozedur «kleiner» zum Vergleich zweier Strings ist nicht ganz universell einsetzbar. Überprüfen Sie bitte, weshalb sie im Programm korrekt arbeitet, beim Aufruf von «kleiner(«Holzweg», «Holz»)» beispielsweise ein eventuell falsches Resultat liefert.

Das zeilen- und seitenweise Formatieren des Quelltextes übernimmt die bedingte Anweisung «IF Zeichen=EOL ...» am Ende der WHILE-Anweisung im Hauptprogramm. Beachten Sie, daß die Prozeduren «MarkiereWort» und «NeueSeite» an Ihren Drucker angepaßt werden müssen.

```
MODULE QuelltextLister;

  FROM InOut IMPORT Read, Write, WriteLn, WriteCard, WriteString,
                    OpenInput, CloseInput, OpenOutput, CloseOutput, Done,
                    EOL;

  CONST ZeilenProSeite = 60;
        MaxZeichenZahl = 80;

  VAR Puffer : ARRAY[0..MaxZeichenZahl] OF CHAR;
      Zeichen, Naechstes : CHAR;
      Zustand : (ZeichenLesen, WortLesen, String1, String2, Kommentar);
      ZeilenNr, ZeichenZahl, KommentarTiefe : CARDINAL;

(*****************************************************************************)
(*        Lokales Modul zur gepufferten Eingabe eines Zeichens               *)
(*****************************************************************************)

  MODULE GepuffertesLesen;

    IMPORT Read;
    EXPORT ReadChar, PushBack;

    VAR ZeichenPuffer : CHAR;

    PROCEDURE ReadChar(VAR Zeichen : CHAR);
      BEGIN
        IF ZeichenPuffer=0C
        THEN Read(Zeichen)
        ELSE Zeichen:=ZeichenPuffer; ZeichenPuffer:=0C
        END (* IF *)
      END ReadChar;

    PROCEDURE PushBack(Zeichen : CHAR);
      BEGIN
        ZeichenPuffer:=Zeichen
      END PushBack;
```

```
    BEGIN (* Initialisierung *)
      ZeichenPuffer:=0C
    END GepuffertesLesen;

(*****************************************************************************)
(*        Lokales Modul zur Bestimmung, ob es sich bei einem Bezeichner      *)
(*        um ein Schluesselwort handelt.                                     *)
(*****************************************************************************)

  MODULE ReservierteWoerter;

    EXPORT ReserviertesWort;

    VAR ResWort: ARRAY[1..40] OF ARRAY[0..15] OF CHAR;

    PROCEDURE ReserviertesWort(Bezeichner : ARRAY OF CHAR) : BOOLEAN;
      VAR von, bis, mitte : [1..40];

      PROCEDURE kleiner(X,Y : ARRAY OF CHAR) : BOOLEAN;
        VAR i, Minimum : CARDINAL;
        BEGIN
          IF HIGH(X)<HIGH(Y)
          THEN Minimum:=HIGH(X)
          ELSE Minimum:=HIGH(Y)
          END; (* IF *)
          i:=0;
          WHILE (i<Minimum) AND (X[i]=Y[i]) AND (X[i]<>0C) AND (Y[i]<>0C) DO
            INC(i)
          END;
          RETURN X[i]<Y[i]
        END kleiner;

      BEGIN
        von:=1; bis:=40;
        WHILE von<>bis DO
          mitte:=(von+bis) DIV 2;
          IF kleiner(ResWort[mitte],Bezeichner)
          THEN von:=mitte+1
          ELSE bis:=mitte
          END (* IF *)
        END; (* WHILE *)
        RETURN NOT kleiner(ResWort[von],Bezeichner) AND
               NOT kleiner(Bezeichner,ResWort[von])
      END ReserviertesWort;

    BEGIN
      ResWort[ 1]:="AND";              ResWort[21]:="LOOP";
      ResWort[ 2]:="ARRAY";            ResWort[22]:="MOD";
      ResWort[ 3]:="BEGIN";            ResWort[23]:="MODULE";
      ResWort[ 4]:="BY";               ResWort[24]:="NOT";
      ResWort[ 5]:="CASE";             ResWort[25]:="OF";
      ResWort[ 6]:="CONST";            ResWort[26]:="OR";
      ResWort[ 7]:="DEFINITION";       ResWort[27]:="POINTER";
      ResWort[ 8]:="DIV";              ResWort[28]:="PROCEDURE";
      ResWort[ 9]:="DO";               ResWort[29]:="QUALIFIED";
      ResWort[10]:="ELSE";             ResWort[30]:="RECORD";
      ResWort[11]:="ELSIF";            ResWort[31]:="REPEAT";
      ResWort[12]:="END";              ResWort[32]:="RETURN";
      ResWort[13]:="EXIT";             ResWort[33]:="SET";
      ResWort[14]:="EXPORT";           ResWort[34]:="THEN";
      ResWort[15]:="FOR";              ResWort[35]:="TO";
      ResWort[16]:="FROM";             ResWort[36]:="TYPE";
```

```
    ResWort[17]:="IF";                ResWort[37]:="UNTIL";
    ResWort[18]:="IMPLEMENTATION";    ResWort[38]:="VAR";
    ResWort[19]:="IMPORT";            ResWort[39]:="WHILE";
    ResWort[20]:="IN";                ResWort[40]:="WITH"
  END ReservierteWoerter;

PROCEDURE MarkiereWort(Bezeichner : ARRAY OF CHAR);
  BEGIN
    Write(CHR(27)); Write('(');
    WriteString(Bezeichner);
    Write(CHR(27)); Write(')')
  END MarkiereWort;

PROCEDURE NeueSeite;
  BEGIN
    Write(CHR(12))
  END NeueSeite;

BEGIN (* QuelltextLister *)
  WriteString("Quelltext-Lister"); WriteLn;
  WriteString("----------------"); WriteLn;
  WriteLn;
  OpenInput("MOD"); IF NOT Done THEN WriteString("Keine Datei!"); HALT END;
  OpenOutput("LST");
  ZeilenNr:=0; Zustand:=ZeichenLesen; Zeichen:=EOL;
  WHILE Done DO
    CASE Zustand OF
      ZeichenLesen : CASE Zeichen OF
                       'A'..'Z' : ZeichenZahl:=0;
                                  Puffer[ZeichenZahl]:=Zeichen;
                                  Zustand:=WortLesen
                     | '"'      : Zustand:=String1
                     | "'"      : Zustand:=String2
                     | "("      : ReadChar(Naechstes);
                                  IF Naechstes='*'
                                  THEN
                                    KommentarTiefe:=1;
                                    Zustand:=Kommentar
                                  END;
                                  PushBack(Naechstes)
                   ELSE (* nichts *)
                   END (* CASE *)
    | WortLesen    : IF (Zeichen>='A') AND (Zeichen<='Z')
                     THEN IF ZeichenZahl<MaxZeichenZahl
                          THEN
                            INC(ZeichenZahl);
                            Puffer[ZeichenZahl]:=Zeichen
                          END
                     ELSE (* Wort vollstaendig gelesen *)
                       IF ZeichenZahl<MaxZeichenZahl
                       THEN
                         INC(ZeichenZahl);
                         Puffer[ZeichenZahl]:=0C
                       END;
                       IF ReserviertesWort(Puffer)
                       THEN MarkiereWort(Puffer)
                       ELSE WriteString(Puffer)
                       END;
                       Zustand:=ZeichenLesen
                     END
    | String1      : IF Zeichen='"' THEN Zustand:=ZeichenLesen END
    | String2      : IF Zeichen="'" THEN Zustand:=ZeichenLesen END
    | Kommentar    : CASE Zeichen OF
```

```
                        '(' : ReadChar(Naechstes);
                              IF Naechstes='*'
                              THEN INC(KommentarTiefe)
                              END;
                              PushBack(Naechstes)
                      | '*' : ReadChar(Naechstes);
                              IF Naechstes=')'
                              THEN
                                DEC(KommentarTiefe);
                                IF KommentarTiefe=0
                                THEN Zustand:=ZeichenLesen
                                END
                              END;
                              PushBack(Naechstes)
                    END (* CASE *)
    END; (* CASE *)
    IF Zeichen=EOL
    THEN
      IF ZeilenNr>0
      THEN IF ZeilenNr MOD ZeilenProSeite = 0
           THEN NeueSeite
           ELSE WriteLn
           END
      END;
      INC(ZeilenNr); WriteCard(ZeilenNr,4); Write(':')
    ELSIF Zustand<>WortLesen
    THEN Write(Zeichen)
    END;
    ReadChar(Zeichen)
  END; (* WHILE *)
  CloseInput;
  CloseOutput;
END QuelltextLister.
```

Aufgabe:

Eine noch bessere Lesbarkeit ergibt sich, wenn auch die Kommentare optisch vom Programmtext getrennt, also beispielweise kursiv ausgedruckt werden. Welche Änderungen sind dafür notwendig?

7.5 Der Datentyp RECORD

Rückblick: Mit Feldern (ARRAY) ist es möglich, in einer Variablen mehrere Daten gleichen Typs zu speichern.

In der Praxis kommt es jedoch häufig vor, daß eine Informationseinheit aus mehreren Daten verschiedenen Typs benötigt wird. Das Standardbeispiel hierfür ist eine Personal-Karteikarte. Hier müssen Angaben wie Namen, Straße und Wohnort (STRING) ebenso eingetragen werden wie Personalnummer (CARDINAL) oder Gehalt (REAL).

Auch ein Datentyp «Datum» kann durch ein Feld nur unzureichend dargestellt werden. Der Versuch

TYPE Datum = ARRAY[1..3] OF CARDINAL;

VAR Heute : Datum;

bei dem die erste Feldkomponente den Tag, die zweite den Monat und die dritte das Jahr enthält, ist unbefriedigend:

- ☐ Hier muß die Interpretation bekannt sein (Dokumentation).
- ☐ Falsche Zuordnungen wie Heute[1]:=1987 werden nicht erkannt.
- ☐ Ein Programm, das diesen Typ benutzt, ist nicht selbstdokumentierend.

Ein zweiter Versuch, das Problem mit einem Feld zu lösen, könnte so aussehen:

```
TYPE  DatumsEintrag = (Tag, Monat, Jahr);
      Datum = ARRAY DatumsEintrag OF CARDINAL;

VAR  Heute : Datum;
```

Hier sind zwar aussagekräftige Zuordnungen möglich:

Heute[Tag]:=27; Heute[Monat]:=11; Heute[Jahr]:=1987;

Dennoch bleibt das Problem der unerkannten Fehlzuordnungen.

Wenn jedoch beispielsweise der Wochentag im Klartext hinzugenommen werden soll, scheitert der Ansatz vollkommen. Für solche Fälle bietet Modula-2 den Datentyp «RECORD». Die einfache Form sieht folgendermaßen aus:

Rekordtyp::="RECORD" Datensatzliste {";"Datensatzliste}
"END".
Datensatzliste::=Feldnamenliste ":" Typ.
Feldnamenliste::=Feldname {"," Feldname}.

Für das obige Beispiel lautet die Deklaration:

```
TYPE Datum = RECORD
               Tag : [1..31];
               Monat : [1..12];
               Jahr : [1900..2100]
             END;
```

Oder mit «Wochentag» im Klartext:

```
TYPE Datum = RECORD
               Wochentag :ARRAY[0..20] OF CHAR;
               Tag : [1..31];
               Monat : [1..12];
               Jahr : [1900..2100]
             END;
```

Auf die einzelnen RECORD-Komponenten wird nicht über einen Index (wie bei ARRAY), sondern über den Feldnamen zugegriffen. Dazu wird dieser vom Variablennamen durch einen Punkt abgetrennt.

Beispiel:

```
VAR Heute : Datum;
...
Heute.Wochentag:="Donnerstag";
Heute.Tag:=14;
Heute.Monat:=5;
Heute.Jahr:=1987;
...
```

7.5.1 Die WITH-Anweisung

Wenn in einer Anweisungssequenz mehrmals auf verschiedene Felder einer RECORD-Variablen zugegriffen werden muß, so kann hier die WITH-Anweisung eingesetzt werden.

WITH-Anweisung::="WITH" (RECORD-)Variable "DO" Anweisungssequenz "END".

Innerhalb der WITH-Anweisung kann dann der Name der RECORD-Variablen einschließlich des folgenden Punktes weggelassen werden.

Beispiel:

```
WITH Heute DO
  Wochentag:="Donnerstag";
  Tag:=31;
  Monat:=5;
  Jahr:=1987
END; (* WITH *)
```

Die WITH-Anweisung kann besonders bei verschachtelten RECORDs vorteilhaft eingesetzt werden.

Beispiel:

```
TYPE Datum = (siehe oben);
     Wetterbeobachtung = RECORD
                           Beobachtungsdatum : Datum;
                           Ort : ARRAY[0..40] OF CHAR;
                           Temperatur : integer;
                           Windrichtung : (Nord, Sued, Ost, West);
                           Windstaerke : 0..7
                         END;
```

VAR HeutigesWetter : Wetterbeobachtung;

Ohne die WITH-Anweisung würde die Datumsangabe folgendermaßen aussehen:

HeutigesWetter.Beobachtungsdatum.Wochentag:="Sonntag";
HeutigesWetter.Beobachtungsdatum.Tag:=19;
HeutigesWetter.Beobachtungsdatum.Monat:=3;
HeutigesWetter.Beobachtungsdatum.Jahr:=1987;

Entsprechend den verschachtelten RECORDs können auch WITH-Anweisungen verschachtelt werden:

```
WITH HeutigesWetter DO
  WITH Datum DO
    Wochentag:="Sonntag";
     Tag:=19; Monat:=3; Jahr:=1987
  END; (* WITH Datum *)
END; (* WITH HeutigesWetter *)
```

Hinweis: Der Einsatz der WITH-Anweisung spart nicht nur Schreibarbeit, sondern führt zudem zu übersichtlicheren, klareren und schnelleren Programmen.

7.5.2 Operationen mit RECORDs

Zwei Variable vom gleichen RECORD-Typ können mit der Zuweisung (:=) gleichgesetzt werden.

Beispiel:

VAR HeutigesWetter, GestrigesWetter : Wetterbeobachtung;

...

HeutigesWetter:=GestrigesWetter;

Darüber hinaus gibt es für RECORDs keine Operatoren. Insbesondere können RECORD-Variable nicht auf Gleichheit oder Ungleichheit geprüft werden (mit = oder <>). Hier müssen vom Programmierer eigene Funktionsprozeduren erstellt werden.

7.5.3 Interaktive Ein- und Ausgabe von RECORDs

Die Bibliotheksmoduln liefern nur Prozeduren zur Ein- und Ausgabe von Daten der Grundtypen CARDINAL, INTEGER, CHAR, REAL usw. Um auch Daten vom RECORD-Typ von der Tastatur ein- bzw. über den Bildschirm auszugeben, müssen eigene Prozeduren geschrieben werden. Hierbei ist auf folgendes zu achten:

- Sicherheit – Fehleingaben müssen erkannt und zurückgewiesen werden.
- Benutzerfreundlichkeit – dem Anwender muß klar sein,was einzugeben ist. Wenn möglich, sollen die Eingaben durch Auswahl erfolgen.

Beispiel: Datumseingabe

Die Ein- und Ausgabe eines Rekordtyps soll an dem häufig benötigten Typ «Datum» demonstriert werden.

```
TYPE Datum = RECORD
               Tag : [1..31];
               Monat : [1..12];
               Jahr : [1800..2100]
             END;
```

Wir wollen die Eingabe eines Datums der Form «19.5.1988» realisieren. Dabei sollen keine unzulässigen Eingaben akzeptiert werden. Der Algorithmus könnte etwa so lauten:

Wiederhole:

Hole eine Datumseingabe von der Tastatur.
Wandle die Datumseingabe in eine Variable vom Typ Datum.
Wenn dabei ein Fehler aufgetreten ist, so gib eine Meldung aus.
Bis die Umwandlung fehlerfrei ist.

Wir lesen also die Tastatureingabe zunächst in eine Zeichenkette und wandeln diese, falls möglich, in ein gültiges Datum um. Aus diesem Grund formulieren wir jetzt die Umwandlungsprozedur

«StringToDatum». Dazu basteln wir uns wieder einen kleinen Automaten mit den Zuständen: TagLesen, MonatLesen, JahrLesen. Der übergebene String wird von links nach rechts zeichenweise analysiert.

alter Zustand	gel	Aktion	neuer Zustand
TagLesen	Ziffer	Tag berechnen	Taglesen
TagLesen	'.'	Tag testen	MonatLesen
TagLesen	sonst	Fehler	-
MonatLesen	Ziffer	Monat berechnen	MonatLesen
MonatLesen	'.'	Monat testen	JahrLesen
MonatLesen	sonst	Fehler	-
JahrLesen	Ziffer	Jahr berechnen	JahrLesen
JahrLesen	sonst	Fehler	-

Es wird kein eigener Ende-Zustand benötigt, da dieser durch das Stringende automatisch angezeigt wird. Am Ende muß das eingegebene Datum noch als ganzes auf Plausibilität geprüft werden.

```
PROCEDURE StringToDatum(S : ARRAY OF CHAR; VAR D : Datum;
                        VAR Fehler : BOOLEAN);

  PROCEDURE BerechneZahl(VAR Zahl : CARDINAL; C : CHAR;
                         VAR Ueberlauf : BOOLEAN);
    BEGIN
      IF Zahl<=(MAX(CARDINAL)-(ORD(C)-ORD('0'))) DIV 10
      THEN Zahl:=10*Zahl + ORD(C)-ORD('0')
      ELSE Ueberlauf:=TRUE
      END (* IF *)
    END BerechneZahl;

  PROCEDURE PruefeDatum(D : Datum; VAR illegal : BOOLEAN);
    BEGIN
      WITH D DO
        CASE Monat OF
          2 : IF (Tag>29) THEN illegal:=TRUE
              ELSIF ((Jahr MOD 4<>0) OR ((Jahr MOD 100=0)
                    AND (Jahr MOD 400<>0))) AND (Tag=29)
              THEN illegal:=TRUE
              ELSE illegal:=FALSE
              END;
        | 4,6,9,11 : illegal:=Tag>30
        ELSE illegal:=FALSE
      END (* CASE *)
    END (* WITH *)
  END PruefeDatum;

  VAR i, Zahl : CARDINAL;
      Zustand : (TagLesen, MonatLesen, JahrLesen);
```

```
BEGIN
  Zahl:=0; Zustand:=TagLesen; Fehler:=FALSE; i:=0;
  WHILE (i<=HIGH(S)) AND (S[i]<>0C) AND NOT Fehler DO
    CASE Zustand OF
      TagLesen : CASE S[i] OF
                   '0'..'9' : BerechneZahl(Zahl,S[i],Fehler)
                 | '.'      : IF (Zahl>0) AND (Zahl<=31)
                              THEN
                                D.Tag:=Zahl;
                                Zahl:=0;
                                Zustand:=MonatLesen
                              ELSE Fehler:=TRUE
                              END (* IF *)
                 ELSE Fehler:=TRUE
                 END (* CASE S[i] OF *)
    | MonatLesen : CASE S[i] OF
                   '0'..'9' : BerechneZahl(Zahl,S[i],Fehler)
                   | '.'      : IF (Zahl>0) AND (Zahl<=12)
                                THEN
                                  D.Monat:=Zahl;
                                  Zahl:=0;
                                  Zustand:=JahrLesen
                                ELSE Fehler:=TRUE
                                END (* IF *)
                   ELSE Fehler:=TRUE
                   END (* CASE S[i] OF *)
    | JahrLesen  : CASE S[i] OF
                   '0'..'9' : BerechneZahl(Zahl,S[i],Fehler)
                   ELSE Fehler:=TRUE
                   END (* CASE S[i] OF *)
    END; (* CASE *)
    INC(i)
  END; (* WHILE *)
  IF Zustand<>JahrLesen THEN Fehler:=TRUE END;
  IF Fehler THEN RETURN END;
  IF (Zahl>=1800) AND (Zahl<=2100)
  THEN D.Jahr:=Zahl;
  ELSE Fehler:=TRUE; RETURN
  END; (* IF *)
  PruefeDatum(D,Fehler)
END StringToDatum;
```

Die Prozedur «LiesDatum» kann nun folgendermaßen formuliert werden:

```
PROCEDURE LiesDatum(VAR D : Datum);
  VAR Fehler : BOOLEAN;
      Eingabe : ARRAY[0..10] OF CHAR;

  PROCEDURE StringToDatum wie oben

   BEGIN (* LiesDatum *)
     REPEAT
       ReadString(Eingabe);
       StringToDatum(Eingabe,D,Fehler);
       IF Fehler THEN WriteString(" ???"); WriteLn END
     UNTIL NOT Fehler
   END LiesDatum;
```

Wesentlich einfacher als die Eingabe gestalten sich in den meisten Fällen die entsprechenden Ausgabeprozeduren, da hier keine Plausibilitätsprüfungen vorgenommen werden müssen. Unsere Prozedur «SchreibDatum» soll allerdings – wie mittlerweile gebräuchlich – führende Nullen bei Tag- und Monatszahlen mit ausgeben «13.09.1987».

```
PROCEDURE SchreibDatum(D : Datum);
  BEGIN
    WITH D DO
      Write(CHR(ORD('0')+Tag DIV 10));
      Write(CHR(ORD('0')+Tag MOD 10));
      Write('.');
      Write(CHR(ORD('0')+Monat DIV 10));
      Write(CHR(ORD('0')+Monat MOD 10));
      Write('.');
      WriteCard(Jahr,4)
    END (* WITH *)
  END SchreibDatum;
```

Und schließlich noch ein kleines Programm zum Testen der Prozeduren:

```
MODULE DatumEinAusgabe;

  FROM InOut IMPORT ReadString, WriteString, WriteLn, Write, WriteCard;

  TYPE Datum = RECORD
                 Tag : [1..31];
                 Monat : [1..12];
                 Jahr : [1800..2100]
               END;

   PROCEDURE LiesDatum wie oben
   PROCEDURE SchreibDatum wie oben

 VAR Heute : Datum;
 BEGIN
   WriteString("Bitte geben Sie das heutige Datum ein: ");
   LiesDatum(Heute);
   WriteLn;
   SchreibDatum(Heute);
   WriteLn
 END DatumEinAusgabe.
```

Probelauf:

Bitte geben Sie das heutige Datum ein: 32.5.1987
???
31,4,2000
???
1.1.1988
01.01.1988

Aufgabe:

Schreiben Sie eine Funktionsprozedur
'FrueherAls(Datum1, Datum2 : Datum):BOOLEAN)',
die genau dann den Wert «TRUE» annimmt, wenn das erste Datum einen Zeitpunkt vor dem zweiten beschreibt.

7.5.4 Ein Modul zum Bruchrechnen

Ein weiteres Beispiel stammt aus der Mathematik. Die Arbeit mit REAL-Zahlen ist relativ ungenau, da Rundungsfehler nicht ausgeschlossen werden können. Häufig werden aber REAL-Zahlen ausschließlich als Dezimalbrüche eingesetzt und nicht zur Annäherung irrationaler Zahlen. Aus diesem Grund wäre es oftmals wünschenswert, einen Datentyp zu haben, der den rationalen Zahlen (Brüche) entspricht. Rechnungen mit Brüchen sind immer genau (solange der zulässige Zahlenbereich nicht überschritten wird).

In Modula-2 kann der Typ «Bruch» ohne weiteres selbst definiert werden:

```
TYPE NatuerlicheZahl = CARDINAL;
     Bruch = RECORD
               Vorzeichen : [-1..+1];
               Zaehler, Nenner : NatuerlicheZahl
             END;
```

Bei dieser Definition wurde absichtlich der Typ «NatuerlicheZahl» anstelle von «CARDINAL» gewählt. Manche Modula-2-Systeme bieten neben «CARDINAL» oft zusätzlich die Typen «LONGCARD» und «LONGINT», die einen wesentlich größeren Zahlenbereich umfassen. Sollte einer dieser Typen auf Ihrem System verfügbar sein, so müssen Sie nur die Typdefinition für «NatuerlicheZahl» entsprechend ändern, um auch den Bereich der Brüche zu erweitern.

Für den Typ «Bruch» müssen nun noch die benötigten Rechenoperationen sowie Möglichkeiten zur Ein- und Ausgabe bereitgestellt werden. Dafür bietet sich selbstverständlich wieder ein eigenes Bibliotheksmodul an, um auf diese Weise das gesamte Modula-2-System an dem neuen Typ teilhaben zu lassen.

```
DEFINITION MODULE Brueche;

(* Ein kleines Modul zum Bruchrechnen. Die Prozeduren
    'StringToBruch' und 'LiesBruch' akzeptieren folgende Darstellung:
```

```
    Bruch::=NormalBruch | DezimalBruch.
    NormalBruch::=["+"|"-"]Zaehler["/"Nenner].
    DezimalBruch::=["+"|"-"]VorkommaZahl["."[Nachkommazahl]].
    Zaehler, Nenner, VorkommaZahl, NachkommaZahl ::= Ziffernfolge.

  Die Variable 'Okay' zeigt eine gültige Umwandlung bzw. Eingabe an *)

TYPE NatuerlicheZahl : CARDINAL; (* bzw. LONGCARD *)
     Bruch = RECORD
               Vorzeichen : [-1..+1];
               Zaehler, Nenner : NatuerlicheZahl
             END;

VAR Okay : BOOLEAN;

PROCEDURE AddiereBruch(X,Y : Bruch; VAR Ergebnis : Bruch);
PROCEDURE SubtrahiereBruch(X,Y : Bruch; VAR Ergebnis : Bruch);
PROCEDURE MultipliziereBruch(X,Y : Bruch; VAR Ergebnis : Bruch);
PROCEDURE DividiereBruch(X,Y : Bruch; VAR Ergebnis : Bruch);
PROCEDURE KuerzeBruch(VAR Ergebnis : Bruch);

PROCEDURE StringToBruch(S : ARRAY OF CHAR; VAR B : Bruch);
PROCEDURE LiesBruch(VAR Ergebnis : Bruch);
PROCEDURE SchreibBruch(B : Bruch);
PROCEDURE SchreibBruchDezimal(B : Bruch; Feld, Stellen : CARDINAL);

END Brueche.
```

Zum Verständnis des Implementationsteils benötigt man nur etwas Schulwissen zum Bruchrechnen. Eine besondere Rolle spielen dabei die Funktionen «ggT» (größter gemeinsamer Teiler) und «kgV» (kleinstes gemeinsames Vielfaches), auf die die Operationen «KuerzeBruch» und «AddiereBruch» zurückgreifen. Für den «ggT» gilt folgender Algorithmus:

Wenn die größere Zahl durch die kleinere ohne Rest teilbar ist,
dann ist die kleinere Zahl der ggT,
andernfalls ist die größere Zahl durch den Rest der Teilung zu ersetzen
und das Verfahren zu wiederholen.

Ein Sonderfall ist zu beachten, wenn eine der (oder gar beide) Zahlen den Wert 0 hat. In diesem Fall ist der «ggT» nicht definiert.

Da die obige Beschreibung noch zu ungenau ist und ohne Rekursion (folgt erst im nächsten Kapitel) nicht unmittelbar in eine Modula-2-Anweisung übersetzt werden kann, nun noch einmal das Ganze als Wiederholungsanweisung.

Wiederhole
Falls die erste Zahl kleiner als die zweite ist,
dann vertausche die beiden Zahlen,

andernfalls ersetze die erste Zahl durch den Rest der Teilung der ersten durch die zweite Zahl,
bis die erste Zahl den Wert 0 erreicht.
Der ggT ist die zweite Zahl.

Bitte überprüfen Sie, ob die beiden Algorithmen zum selben Resultat führen, indem Sie beide mit ein paar Zahlenpaaren durchspielen.

Das kleinste gemeinsame Vielfache (kgV, gemeinsamer Nenner) kann direkt mit dem ggT bestimmt werden: kgV(X,Y) = X*Y DIV ggT(X,Y).

Der Algorithmus zum vollständigen Kürzen eines Bruches lautet:

ZählerNeu = Zähler DIV ggT(Zähler,Nenner)
NennerNeu = Nenner DIV ggT(Zähler,Nenner).

Frage: Aus welchem Grund wird in «KuerzeBruch» die Variable «temp» benötigt?

Am kompliziertesten ist sicherlich die Addition zweier Brüche, da das Vorzeichen als eigene Größe behandelt werden muß und Ganzzahl-Unterläufe abgefangen werden müssen.

Die interessanteste Prozedur dürfte sich hinter «StringToBruch» verbergen, die wiederum mit einem Automaten realisiert wurde. Diese Prozedur akzeptiert sowohl Brüche der Form «Zähler/Nenner» als auch Dezimalbrüche.

Die Prozeduren zur Aus- und Eingabe von Brüchen wiederum dürften keine Probleme mehr bereiten.

```
IMPLEMENTATION MODULE Brueche;

  FROM InOut IMPORT ReadString, WriteCard, WriteInt, Write;

  PROCEDURE ggT(X,Y : NatuerlicheZahl):NatuerlicheZahl;
    VAR temp : NatuerlicheZahl;
    BEGIN
      IF X*Y=0 THEN RETURN 1 END;
      REPEAT
        IF X<Y THEN temp:=X; X:=Y; Y:=temp ELSE X:=X MOD Y END
      UNTIL X=0;
      RETURN Y
    END ggT;

  PROCEDURE kgV(X,Y : NatuerlicheZahl):NatuerlicheZahl;
    BEGIN
      RETURN X DIV ggT(X,Y) * Y
    END kgV;
```

```
PROCEDURE KuerzeBruch(VAR Ergebnis : Bruch);
  VAR temp : NatuerlicheZahl;
  BEGIN
    WITH Ergebnis DO
      IF Vorzeichen=0 THEN RETURN END;
      temp:=ggT(Zaehler, Nenner);
      Zaehler:=Zaehler DIV temp;
      Nenner:=Nenner DIV temp
    END (* WITH *)
  END KuerzeBruch;

PROCEDURE AddiereBruch(X,Y : Bruch; VAR Ergebnis : Bruch);
  BEGIN
    IF X.Vorzeichen=0 THEN Ergebnis:=Y
    ELSIF Y.Vorzeichen=0 THEN Ergebnis:=X
    END; (* IF *)
    Ergebnis.Nenner:=kgV(X.Nenner,Y.Nenner);
    X.Zaehler:=X.Zaehler*(Ergebnis.Nenner DIV X.Nenner);
    Y.Zaehler:=Y.Zaehler*(Ergebnis.Nenner DIV Y.Nenner);
    IF X.Vorzeichen=-1
    THEN IF Y.Vorzeichen=-1
         THEN
           Ergebnis.Vorzeichen:=-1;
           Ergebnis.Zaehler:=X.Zaehler+Y.Zaehler
         ELSIF X.Zaehler>Y.Zaehler
         THEN
           Ergebnis.Vorzeichen:=-1;
           Ergebnis.Zaehler:=X.Zaehler-Y.Zaehler
         ELSIF X.Zaehler<Y.Zaehler
         THEN
           Ergebnis.Vorzeichen:=+1;
           Ergebnis.Zaehler:=Y.Zaehler-X.Zaehler
         ELSE Ergebnis.Vorzeichen:=0
         END
    ELSE IF Y.Vorzeichen=+1
         THEN
           Ergebnis.Vorzeichen:=+1;
           Ergebnis.Zaehler:=X.Zaehler+Y.Zaehler
         ELSIF X.Zaehler>Y.Zaehler
         THEN
           Ergebnis.Vorzeichen:=+1;
           Ergebnis.Zaehler:=X.Zaehler-Y.Zaehler
         ELSIF X.Zaehler<Y.Zaehler
         THEN
           Ergebnis.Vorzeichen:=-1;
           Ergebnis.Zaehler:=Y.Zaehler-X.Zaehler
         ELSE Ergebnis.Vorzeichen:=0
         END
    END;
    KuerzeBruch(Ergebnis)
  END AddiereBruch;

PROCEDURE SubtrahiereBruch(X,Y : Bruch; VAR Ergebnis : Bruch);
  BEGIN
    Y.Vorzeichen:=-Y.Vorzeichen;
    AddiereBruch(X,Y,Ergebnis)
  END SubtrahiereBruch;
```

```
PROCEDURE MultipliziereBruch(X,Y : Bruch; VAR Ergebnis : Bruch);
  BEGIN
    WITH Ergebnis DO
      Vorzeichen:=X.Vorzeichen*Y.Vorzeichen;
      Zaehler:=X.Zaehler*Y.Zaehler;
      Nenner:=X.Nenner*Y.Nenner
    END; (* WITH *)
    KuerzeBruch(Ergebnis)
  END MultipliziereBruch;

PROCEDURE DividiereBruch(X,Y : Bruch; VAR Ergebnis : Bruch);
  BEGIN
    IF Y.Vorzeichen=0 THEN Ergebnis.Vorzeichen:=0; Okay:=FALSE; RETURN END;
    WITH Ergebnis DO
      Vorzeichen:=X.Vorzeichen*Y.Vorzeichen;
      Zaehler:=X.Zaehler*Y.Nenner;
      Nenner:=X.Nenner*Y.Zaehler
    END; (* WITH *)
    KuerzeBruch(Ergebnis)
  END DividiereBruch;

PROCEDURE StringToBruch(S : ARRAY OF CHAR; VAR B : Bruch);
  VAR i : CARDINAL;
      Zustand :(Anfang, ZaehlerLesen, NennerLesen, Nachkommastellen);
  BEGIN
    Zustand:=Anfang; Okay:=TRUE; i:=0;
    WITH B DO
      Vorzeichen:=+1; Zaehler:=0; Nenner:=0;
      WHILE (i<=HIGH(S)) AND (S[i]<>0C) DO
        CASE S[i] OF
          '-', '+' : IF Zustand<>Anfang
                     THEN Okay:=FALSE; RETURN
                     ELSE
                       Zustand:=ZaehlerLesen;
                       IF S[i]='-' THEN Vorzeichen:=-1 END
                     END
        | ' ' : IF Zustand<>Anfang THEN Okay:=FALSE; RETURN END
        | '/' : IF Zustand<>ZaehlerLesen
                THEN Okay:=FALSE; RETURN
                ELSE Zustand:=NennerLesen
                END
        | '.' : IF Zustand<>ZaehlerLesen
                THEN Okay:=FALSE; RETURN
                ELSE Zustand:=Nachkommastellen; Nenner:=1;
                END
        | '0'..'9' : CASE Zustand OF
                       Anfang : Zustand:=ZaehlerLesen;
                                Zaehler:=ORD(S[i])-ORD('0')
                     | ZaehlerLesen :
                                Zaehler:=10*Zaehler+ORD(S[i])-ORD('0')
                     | NennerLesen :
                                Nenner:=10*Nenner+ORD(S[i])-ORD('0')
                     | Nachkommastellen:
                                Zaehler:=10*Zaehler+ORD(S[i])-ORD('0');
                                Nenner:=10*Nenner
                     END (* CASE *)
        ELSE Okay:=FALSE; RETURN
        END; (* CASE *)
        INC(i)
      END; (* WHILE *)
      IF Nenner=0
```

```
      THEN Nenner:=1; IF Zustand<>ZaehlerLesen THEN Okay:=FALSE END;
      END; (* IF Nenner=0 *)
      IF Zaehler=0 THEN Vorzeichen:=0 END;
    END (* WITH *)
  END StringToBruch;

PROCEDURE LiesBruch(VAR Ergebnis : Bruch);
  VAR S : ARRAY[1..80] OF CHAR;
  BEGIN
    ReadString(S); StringToBruch(S,Ergebnis)
  END LiesBruch;

PROCEDURE SchreibBruch(B : Bruch);
  BEGIN
    IF B.Vorzeichen=-1
    THEN Write('-')
    ELSIF B.Vorzeichen=0
    THEN Write('0'); RETURN
    END; (* IF *)
    WriteCard(B.Zaehler,1);
    IF B.Nenner>1 THEN Write('/'); WriteCard(B.Nenner,1) END
  END SchreibBruch;

PROCEDURE SchreibBruchDezimal(B : Bruch; Feld, Stellen : CARDINAL);
  VAR i, Rest : CARDINAL;
  BEGIN
    WITH B DO
      WriteInt(Vorzeichen*INTEGER(Zaehler DIV Nenner),Feld-Stellen-1);
      Rest:=10*(Zaehler MOD Nenner);
      IF Stellen>0 THEN Write('.') END;
      FOR i:=1 TO Stellen DO
        WriteCard(Rest MOD Nenner,1);
        Rest:=10*(Zaehler MOD Nenner)
      END (* FOR *)
    END (* WITH *)
  END SchreibBruchDezimal;

END Brueche.
```

Schließlich folgt wiederum ein kleines Programm zum Testen des neuen Datentyps.

```
MODULE BruchTest;

  FROM InOut IMPORT Write, WriteString, WriteLn, Read;

  FROM Brueche IMPORT Bruch, LiesBruch, SchreibBruch, AddiereBruch,
                      SubtrahiereBruch, MultipliziereBruch,
                      DividiereBruch, Okay;

  VAR B1, B2, B : Bruch;
      Rechenzeichen, Antwort : CHAR;
      korrekt : BOOLEAN;

  BEGIN
    WriteString("Bruchrechnen"); WriteLn;
    WriteString("------------"); WriteLn;
    WriteLn;
```

```
    WriteString("Bitte geben Sie Aufgaben der Form ");
    WriteString("'Bruch1 Rechenzeichen Bruch2' ein."); WriteLn;
    WriteLn;
    REPEAT
      LiesBruch(B1); korrekt:=Okay;
      Read(Rechenzeichen);
      LiesBruch(B2); korrekt:=korrekt AND Okay;
      CASE Rechenzeichen OF
         '+' : AddiereBruch(B1,B2,B)
       | '-' : SubtrahiereBruch(B1,B2,B)
       | '*' : MultipliziereBruch(B1,B2,B)
       | ':' : DividiereBruch(B1,B2,B); korrekt:=korrekt AND Okay
       ELSE korrekt:=FALSE
      END; (* CASE *)
      IF korrekt
      THEN WriteString(" = "); SchreibBruch(B); WriteLn;
      ELSE WriteString("???"); WriteLn
      END; (* IF *)
      WriteString("Noch eine Aufgabe (J,N) ? ");
      REPEAT
        Read(Antwort); Antwort:=CAP(Antwort)
      UNTIL (Antwort='J') OR (Antwort='N');
      WriteLn
    UNTIL Antwort='N'
  END BruchTest.
```

Probelauf:

Bruchrechnen

Bitte geben Sie Aufgabe der Form 'Bruch1 Rechenzeichen Bruch2' ein.

−1.2 + 3/4 = −9/20
Noch eine Aufgabe (J,N) ? J
4/3 * 3/4 = 1
Noch eine Aufgabe (J,N) ? J
1 : 0.125 = 8
Noch eine Aufgabe (J,N) ?

Aufgaben:

1. Bestimmen Sie die Grenzen des Typs Bruch, also den größten und kleinsten positiven (von Null verschiedenen) Bruch.
2. Erweitern Sie das Modul «Brueche» um eine Vergleichsfunktion «BruchKleiner», mit der geprüft werden kann, ob ein Bruch kleiner als ein anderer ist.

7.5.5 RECORD mit strukturierten Komponenten

Ausgehend von den einfachen Datentypen können mit RECORD- und ARRAY-Typen beliebig zusammengesetzte Datenstrukturen angelegt werden. Das Finden der richtigen, d.h. dem Problem angemessenen Datenstruktur gehört zu den wichtigsten Programmierarbeiten. Der Erfinder von Modula-2, der Schweizer Professor Niklaus Wirth, stellte in einem Buchtitel folgende Gleichung auf:

Datenstrukturen + Algorithmen = Programme

Beispiel:

In einem Programm sollen Meßreihen bearbeitet werden. Dazu werden von den einzelnen Meßstationen folgende Daten gesendet:

Datum und Uhrzeit der Messung, Name des Meßleiters, 400 Meßwertpaare.

Um die einzelnen Meßreihen später wieder den Stationen zuordnen zu können, müssen diese auch mit aufgenommen werden. Eine adäquate Datenstruktur könnte etwa so aussehen:

```
CONST   AnzahlMesswerte = 400;

TYPE    Datum = RECORD
                  Tag : [1..31];
                  Monat : [1..12];
                  Jahr : [1900..2100]
                END;

        Zeit =  RECORD
                  Stunde : [0..23];
                  Minute, Sekunde : [0..59]
                END;

        MessStation = [1..10]; (* 10 Stationen per Nr. identifiziert *)

        MesswertPaar = RECORD
                         x, y : REAL
                       END;

        Messwerte = ARRAY[1..AnzahlMesswerte] OF MesswertPaar;

        Name = RECORD
                 Familienname, Vorname : ARRAY[0..30] OF CHAR
               END;

        MessReihe = RECORD
                      Station : MessStation;
                      DatumDerMessung : Datum;
                      ZeitDerMessung : Zeit;
                      Messleiter : Name;
                      Messung : Messwerte
                    END;
```

Die letzte Definition «MessReihe» ist sehr abstrakt. Hier kommen überhaupt keine Grundtypen mehr vor. Dennoch entspricht sie exakt den (fiktiven) Vorgaben.

Aufgaben:

1. Definieren Sie einen Datentyp Adresse (Name, Vorname, Straße, Postleitzahl, Ort, Telefon). Schreiben Sie hierfür eine Eingabeprozedur sowie zwei Ausgabeprozeduren, die a) einen ganzen Datensatz und b) nur Name, Vorname und Telefon ausgeben.
2. Schreiben Sie, aufbauend auf Aufgabe 1, eine kleine Adreßverwaltung für max. 100 Adressen. Fügen Sie eine Prozedur ein, die Ihre Adressen nach Namen und Vornamen sortiert (Quicksort). Ermöglichen Sie die Suche einer Adresse durch binäre Suche im sortierten Feld.
3. Schreiben Sie die Prozeduren «LiesZeit» und «SchreibZeit» zur Ein- und Ausgabe von Variablen des Typs «Zeit» (22:01:14).

7.5.6 RECORD mit Varianten

Trotz der großen Datenvielfalt, die uns mittels RECORD-Konstruktionen zur Verfügung stehen, gibt es Fälle, in denen die bisher aufgezeigten Möglichkeiten nicht ausreichen. Dazu wieder ein Beispiel aus der Praxis der Personenkarteien:

In einer (fiktiven) Firma gibt es drei Beschäftigungskategorien.

Teilzeit:	Der Arbeitnehmer enthält einen Stundenlohn.
Fest:	Der Arbeitnehmer erhält ein festes Monatsgehalt.
Auftrag:	Der Arbeitnehmer erhält eine prozentuale Gewinnbeteiligung.

Mit den bisher bekannten Möglichkeiten müßte ein Typ «Beschäftigung» so aussehen:

```
TYPE Beschaeftigung= RECORD
                       Art : (Teilzeit, Fest, Auftrag);
                       Stundenzahl : [0..200];
                       Stundenlohn : REAL;
                       Monatslohn : REAL;
                       Ergebnis : REAL;
                       Prozentsatz : [10..20]
                     END;
```

Dieser Ansatz ist unbefriedigend, da bei einer Teilzeitkraft die Einträge für «Monatslohn», «AuftragsErgebnis» und «Prozentsatz»

grundsätzlich leer bleiben und nicht relevant sind. Hier wäre eine Fallunterscheidung schon bei der Anlage des Datentyps sinnvoll. Genau diese Fallunterscheidung kann mit varianten RECORDs realisiert werden. Dazu die vollständige RECORD-Definition:

```
Rekordtyp::="RECORD" Datensatzliste (";" Datensatzliste) "END".
Datensatzliste::=[FesterTeil | VarianterTeil].
FesterTeil::=Feldnamenliste ":" Typ.
Feldnamenliste::=Feldname ("," Feldname ).
VarianterTeil::="CASE" [Auswahlfeld] ":" Auswahltyp "OF"
                Variante ("|" Variante) ["ELSE" Datensatzliste]
                "END".
Auswahlfeld::=Bezeichner.
Auswahltyp::=Aufzählungstyp | CARDINAL | INTEGER |
             BOOLEAN | CHAR | Unterbereichstyp.
Variante::=Markenliste ":" Datensatzliste.
Markenliste::=Marke ("," Marke).
Marke::=Konstante | Konstante ".." Konstante
```

Wichtig an dieser Definition sind folgende Besonderheiten:

- ☐ Es können beliebig viele variante Teile enthalten sein.
- ☐ Ein RECORD-Typ kann auch ausschließlich aus einem varianten Teil bestehen.
- ☐ Das Auswahlfeld (tag-field) kann auch entfallen.
- ☐ Jede Variante hat ein eigenes «END».
- ☐ Das Auswahlfeld ist eine eigene RECORD-Komponente.
- ☐ Wie bei der CASE-Anweisung dürfen statt der Konstanten auch konstante Ausdrücke benutzt werden.

Mit dieser Erweiterung ist das obige Problem leicht zu lösen:

```
TYPE BeschArt = (Teilzeit, Fest, Auftrag);
     Beschaeftigung = RECORD
                        CASE Kathegorie : BeschArt OF
                           Teilzeit : Stundenzahl : [0..200];
                                      Stundenlohn : REAL;
                         | Fest     : Monatslohn : REAL;
                         | Auftrag  : Ergebnis : REAL;
                                      Prozentsatz : [10..20]
                        END (* CASE *)
                      END; (* RECORD *)
```

Der Zugriff auf die einzelnen Komponenten erfolgt in der bekannten Weise:

```
VAR Angestellter : Beschaeftigung;
...
Angestellter.Kategorie:=Fest;
Angestellter.Monatslohn:=3500.00
```

Achtung: Die Zugriffe auf die einzelnen Varianten werden weder vom Compiler noch vom Laufzeitsystem überprüft! Es sind also auch folgende Anweisungen möglich:

```
Angestellter.Kategorie:=Teilzeit;
Angestellter.Prozentsatz:=12;
Angestellter.Monatslohn:=3500.00;
```

Hier werden keine Fehler gemeldet. Bei der Arbeit mit varianten Records liegt die Verantwortung voll in den Händen des Programmierers.

7.5.7 Die CASE-Anweisung und variante Records

Der sicherste Weg liegt in der Verwendung der CASE-Anweisung:

```
CASE Angestellter.Kathegorie OF
   Teilzeit : Angestellter.Stundenlohn:=24.50;
 | Fest     : Angestellter.Monatslohn:=3500.00;
 | Auftrag  : Angestellter.Prozentsatz:=12
END;
```

Oder mit der WITH-Anweisung:

```
WITH Angestellter DO
  CASE Kathegorie OF
     Teilzeit : Stundenlohn:=24.50;
   | Fest     : Monatslohn:=3500.00;
   | Auftrag  : Prozentsatz:=12
  END
END;
```

Regel:

Die CASE-Anweisung ist adäquate Anweisungsstruktur zur Datenstruktur varianter Records.

Bei RECORDs mit Varianten wird sich die CASE-Anweisung in allen Ein- und Ausgabeprozeduren wiederfinden.

Aufgabe:

Erweitern Sie Ihr Adreßprogramm um die Angabe eines Steckenpferdes (Tennis, Musik, Computer). Dabei sollen – je nach Hobby – noch weitere Informationen gespeichert werden:

Tennis:	Spielstärke (stark, mittel, schwach);
Musik:	Instrument (Gitarre, Orgel, Flöte) und Virtuosität (gut, schlecht)

Computer: Programmiersprache (Pascal, BASIC, Prolog) und Betriebssystem (CP/M, MS-DOS)

Erweitern Sie Ihre Ein- und Ausgabeprozeduren entsprechend.

7.5.8 Programmiertips

Abschließend zu diesem Thema noch ein paar Tips zur Programmierung von interaktiven Eingaberoutinen für RECORDs:

- ☐ Schreiben Sie erst komfortable Eingabeprozeduren für die Basistypen der einzelnen Komponenten.
- ☐ Benutzen Sie eigene Routinen für Zahleneingaben.
- ☐ Geben Sie alle benötigten Hinweise für den Anwender.
- ☐ Prüfen Sie die Eingaben auf Plausibilität und Zulässigkeit.
- ☐ Setzen Sie die Eingabeprozedur für komplexe Records aus den Eingabeprozeduren für die Basistypen zusammen.
- ☐ Versuchen Sie möglichst, Records über Bildschirmmasken einzugeben.

7.6 Prozeduren als Datentyp

Auch ganze Prozeduren können als Datentyp aufgefaßt werden:

Prozedur-Typ::="PROCEDURE" [Formale-Typenliste].
Formale-Typenliste::= "(" [["VAR"] Typ] {"," ["VAR"] Typ} ")"[":" Typ].

Für parameterlose Prozeduren gibt es den Standardtyp «PROC».

Beispiele:

```
TYPE    Eingabeprozedur = PROCEDURE(CHAR);
        Zeichenumwandlung = PROCEDURE(CHAR,VAR
        CHAR);
        RealeFunktion = PROCEDURE(REAL) : REAL;
        BildschirmAktion = PROC;
```

Damit ist ein sehr einfacher Weg gegeben, Prozeduren als Parameter an andere Prozeduren zu übergeben. Allerdings ist darauf zu achten, daß Variable eines Prozedurtyps keine Standardprozeduren zugewiesen werden dürfen.

Als Beispiel wollen wir nochmals das Sortieren eines Zahlenfeldes behandeln. Um die einzelnen Prozeduren des Programms möglichst universell zu gestalten, benutzen wir auch diesmal offene Felder und zusätzlich eine Vergleichsfunktion als Parameter. Dadurch kann der Prozedur «Sortierung» mitgeteilt werden, ob das übergebene Feld auf- oder absteigend geordnet werden soll.

```
MODULE Zahlenfelder2;

  FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;

  TYPE  VergleichsFunktion = PROCEDURE(CARDINAL,CARDINAL):BOOLEAN;

  CONST MaxZahlen = 100;

  VAR   Feld : ARRAY[1..MaxZahlen] OF CARDINAL;
        Anzahl : CARDINAL;

  PROCEDURE Eingabe(VAR F : ARRAY OF CARDINAL; VAR A : CARDINAL);
    VAR NaechsteZahl : CARDINAL;
    BEGIN
      A:=0;
      WriteString("Bitte geben Sie eine Folge von max. ");
      WriteCard(MaxZahlen,1);
      WriteString(" CARDINAL-Zahlen ein."); WriteLn;
      WriteString("Ende der Zahlenfolge = 0"); WriteLn;
      ReadCard(NaechsteZahl);
      WHILE (A<=HIGH(F)) AND (NaechsteZahl>0) DO
        F[A]:=NaechsteZahl;
        ReadCard(NaechsteZahl);
        INC(A)
      END; (* WHILE *)
      WriteLn
    END Eingabe;

  PROCEDURE Ausgabe(F : ARRAY OF CARDINAL; A : CARDINAL);
    CONST MaxSpalten = 10; (* 80-Zeichen-Bildschirm *)
          BildschirmZeilen = 24;
    VAR i, j : CARDINAL;
        ZeilenZahl : CARDINAL;
        SpaltenZahl : [1..MaxSpalten];
    BEGIN
      IF (A+BildschirmZeilen-1) DIV BildschirmZeilen>MaxSpalten
      THEN SpaltenZahl:=MaxSpalten
      ELSE SpaltenZahl:=(A+BildschirmZeilen-1) DIV BildschirmZeilen
      END; (* IF *)
      ZeilenZahl:=(A+SpaltenZahl-1) DIV SpaltenZahl;
      FOR i:=1 TO ZeilenZahl DO
        FOR j:=0 TO SpaltenZahl-1 DO
          IF i+j*ZeilenZahl<=A THEN WriteCard(F[i+j*ZeilenZahl-1],8) END
        END; (* FOR j *)
        IF SpaltenZahl<MaxSpalten THEN WriteLn END
      END; (* FOR i *)
      WriteLn
    END Ausgabe;
```

```
PROCEDURE Sortierung(VAR F : ARRAY OF CARDINAL; A : CARDINAL;
                     V : VergleichsFunktion);
  VAR i, j, kleinstes : CARDINAL;
      temp : CARDINAL;
  BEGIN (* Sortierung *)
    FOR i:=0 TO A-2 DO
      kleinstes:=i;
      FOR j:=i+1 TO A-1 DO
        IF V(F[j],F[kleinstes]) THEN kleinstes:=j END
      END; (* FOR j *)
      IF i<>kleinstes
      THEN temp:=F[i]; F[i]:=F[kleinstes]; F[kleinstes]:=temp
      END
    END (* FOR i *)
  END Sortierung;

PROCEDURE kleiner(X,Y : CARDINAL):BOOLEAN;
  BEGIN
    RETURN X<Y
  END kleiner;

PROCEDURE groesser(X,Y : CARDINAL):BOOLEAN;
  BEGIN
    RETURN X>Y
  END groesser;

BEGIN (* Zahlenfelder2 *)
  Eingabe(Feld,Anzahl);
  WriteString("Aufsteigende Reihenfolge:"); WriteLn;
  Sortierung(Feld,Anzahl,kleiner);
  Ausgabe(Feld,Anzahl);
  WriteString('Absteigende Reihenfolge: '); WriteLn;
  Sortierung(Feld,Anzahl,groesser);
  Ausgabe(Feld,Anzahl);
END Zahlenfelder2.
```

Probelauf:

Bitte geben Sie eine Folge von max. 100 CARDINAL-Zahlen ein.
Ende der Zahlenfolge = 0

1
23
35
22
16
4
0

Aufsteigende Reihenfolge:	Absteigende Reihenfolge
1	35
4	23
16	22
22	16
23	4
35	1

8 Rekursion

Die erste höhere Programmiersprache war FORTRAN (FORmula TRANslator, Formelübersetzer). Sie wird nach wie vor im technisch-naturwissenschaftlichen Bereich eingesetzt. Daß diese Programmiersprache Rekursion ausdrücklich verbietet, war einer der Hauptgründe für die Schaffung neuer Programmiersprachen wie ALGOL, Pascal und schließlich Modula-2. «Erst wer die Rekursion beherrscht, kann richtig programmieren» – diesen Satz hören Informatik-Studenten bereits in der ersten Vorlesungsstunde. Wahr daran ist auf jeden Fall, daß die Verfügbarkeit rekursiver Algorithmen und rekursiver Datenstrukturen dem Programmierer ein mächtiges Werkzeug in die Hand gibt, dessen Beherrschung viel zur Lösung schwieriger Programmierprobleme beiträgt. In diesem Kapitel geht es um ein grundlegendes Verständnis der Rekursion.

8.1 Rekursive Objekte

Objekte heißen rekursiv, wenn sie sich selbst enthalten. Ein Begriff wird rekursiv definiert, wenn im Definitionsteil der Begriff selbst wieder verwendet wird.

Ein ganz alltägliches Beispiel für eine rekursive Definition kann man bei Begriffen wie «Nachkomme» oder «Vorfahr» antreffen. Ein erster Ansatz wie

Nachkomme ist ein Kind oder ein Enkel oder ein Urenkel oder ...

führt nicht zum erwünschten Erfolg, da hier nur ein paar Beispiele für «Nachkommen» angegeben werden, das Wesen des Begriffs jedoch nicht erfaßt wird. Auch der zweite Versuch

Nachkomme ist ein Kind oder das Kind eines Kindes oder das Kind eines Kindes eines Kindes usw.

ist noch keine befriedigende Lösung. Allerdings versteckt sich hier schon das Rekursionsprinzip in

- der offensichtlich gleichförmigen Zusammensetzung der einzelnen Nachkommen (Kind, Kind eines Kindes, Kind eines Kindes eines Kindes)
- dem Nachsatz «usw.»

Die rekursive Definition von «Nachkomme» lautet somit:

Nachkomme ist ein Kind oder das Kind eines Nachkommen.

Diese Definition ist nicht nur kürzer als die vorhergehenden, sie ist auch die einzige, die die Bedeutung des Begriffs «Nachkomme» vollständig beinhaltet. Sie erlaubt die Prüfung beliebig langer Abstammungsketten auf Nachkommenschaft. Dazu ein kleines Beispiel:

Hans ist das Kind von Otto.
Petra ist das Kind von Hans.
Hubert ist das Kind von Petra.

Die Frage, ob Hubert ein Nachkomme von Hans ist, läßt sich nach unserer Definition folgendermaßen klären:

Hubert ist ein Nachkomme von Hans,
wenn (1) Hubert ein Kind von Hans ist oder
(2) Hubert das Kind eines Nachkommen von Hans ist.

Da (1) nicht zutrifft, muß geprüft werden, ob Hubert das Kind eines Nachkommen von Hans ist. Hubert ist das Kind von Petra. Wenn nun gezeigt werden kann, daß Petra ein Nachkomme von Hans ist, so gilt diese Beziehung auch für Hubert.

Petra ist ein Nachkomme von Hans,
wenn (1) Petra ein Kind von Hans ist oder
(2) Petra das Kind eines Nachfolgers von Hans ist.

Hier trifft (1) zu. Die Prüfung ist abgeschlossen, der Nachweis ist erbracht. Andere interessante rekursive Definitionen:

Jeder Nachfolger einer natürlichen Zahl ist eine natürliche Zahl.

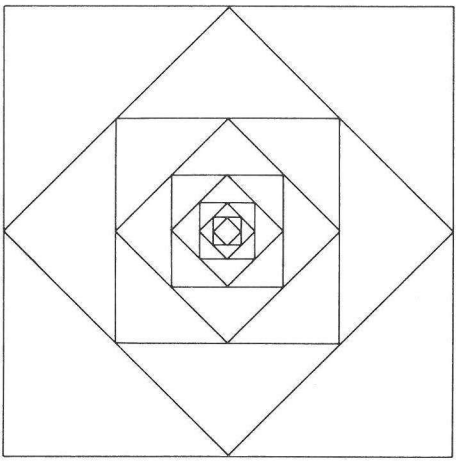
Die Fakultät einer Zahl ist das Produkt aus dieser Zahl und der Fakultät der um eins erniedrigten Zahl.

8.2 Rekursive Figuren

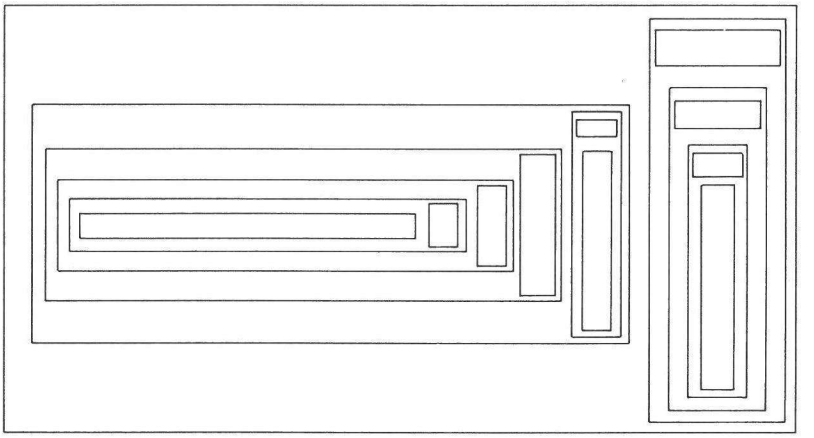
So richtig nachvollziehbar wird das Wesen der Rekursion in rekursiv definierten geometrischen Figuren.

Beispiel:

A ist ein Quadrat, dem ein A so eingeschrieben ist, daß seine Eckpunkte die Seitenmitten der umschließenden Figur sind.



B ist ein Rechteck, das im rechten Teil ein um 90 Grad gedrehtes, verkleinertes B und im linken Drittel ein verkleinertes B enthält.



8.3 Rekursive Prozeduren

Wenn in Modula-2 eine Prozedur deklariert wird, so ist sie bereits nach dem Abschluß des Prozedurkopfes bekannt. Sie kann also in ihrem eigenen Anweisungsteil aufgerufen werden.

```
PROCEDURE A;
...
BEGIN
...
A;
...
END A;
```

Den Selbstaufruf einer Prozedur kann man am besten nachvollziehen, indem man sich dieselbe Prozedur unter anderem Namen nochmals deklariert denkt, die an dieser Stelle aufgerufen wird. Es gilt vor allem uneingeschränkt das Prinzip der Lokalität: Bei jedem Aufruf werden die lokalen Variablen neu zur Verfügung gestellt.

Aus dem bisher Gesagten ergibt sich bereits folgende Konsequenz: Eine Prozedur darf sich höchstens bedingt rekursiv aufrufen, da sonst eine endlose Rekursion vorliegt.

In rekursiven Prozeduren muß immer ein Rekursionsabbruch eingebaut sein.

Rekursive Prozeduren haben also prinzipiell folgende Gestalt:

```
PROCEDURE RekursiverAufruf;

  BEGIN
     IF Bedingung
     THEN RekursiverAufruf
     END
  END RekursiverAufruf;
```

Bisher war immer nur von Prozeduren die Rede, die sich selbst aufrufen. In diesem Fall spricht man von «direkter Rekursion».

Es ist aber auch möglich, daß die Prozedur A eine Prozedur B aufruft und die Prozedur B wiederum die Prozedur A. Diese Art der Rekursion heißt «indirekt». Indirekte Rekursion ist mit Vorsicht zu genießen, da hier die Abbruchkriterien oftmals sehr undurchsichtig sind. Um einen Eindruck von den möglichen Gefahren zu vermitteln, hier nochmals ein Beispiel aus dem Familienleben:

X ist der Vater von Y, wenn Y ein Kind von X ist.
X ist ein Kind von Y, wenn Y der Vater von X ist.

Hier wird Vater durch Kind und Kind durch Vater erklärt, insgesamt aber nichts ausgesagt.Solche Konstruktionen sind in Modula-2 durchaus möglich. Eine Prozedur kann eine andere aufrufen, obwohl diese noch nicht deklariert wurde. Voraussetzung dafür ist nur, daß sie (später) im selben oder einem äußeren Block definiert wird.

Beispiel:

```
PROCEDURE B;
   BEGIN
      ...
      A
   END B;

PROCEDURE A;
  BEGIN
    ...
    B
  END A;
```

Hinweis: Bei einigen älteren Systemen muß der Prozedurkopf einer noch nicht deklarierten Prozedur angegeben werden, bevor sie in einer anderen benutzt werden kann. Dieser Prozedurkopf wird mit dem Schlüsselwort «FORWARD» abgeschlossen.

```
PROCEDURE A; FORWARD; (* Damit wird die Prozedur vor-deklariert *)

PROCEDURE B; (* Hier kann jetzt auf A zugegriffen werden *)
  BEGIN
  ...
  A
  END B;

PROCEDURE A; (* Erst hier wird A vollständig ausgeführt *)
  BEGIN
  ...
  B
  END A;
```

Falls vorhanden, müssen die formalen Parameter bei dem vorwärts deklarierten Prozedurkopf und der späteren Ausführung übereinstimmen.

Ein praktisches Beispiel für den Gebrauch indirekter Rekursion kann man in Programmen finden, die irgendwelche Ausgaben seitenweise formatieren:

```
PROCEDURE DruckeZeile(Zeile : ARRAY OF CHAR);
  BEGIN
    IF KeinPlatzMehrAufDerSeite THEN NeueSeite END;
    WriteString(Zeile)
  END DruckeZeile;

PROCEDURE NeueSeite;
  BEGIN
    Write(SeitenVorschub);
    DruckeZeile(KopfZeile);
    DruckeZeile(Datum);
    DruckeZeile("")
  END NeueSeite;
```

8.4 Die Fakultät

Alles, was bisher über Prozeduren gesagt wurde, gilt natürlich auch für Funktionen!

Als erstes, konkretes Beispiel wollen wir die Fakultät-Funktion in Angriff nehmen. Die exakte mathematische Definition lautet:

Die Fakultät von 0 ist 1.
Die Fakultät einer größeren Zahl als Null ist das Produkt aus dieser Zahl und der Fakultät der um 1 verringerten Zahl.

oder (in mathematischer Schreibweise)

f(x) = 1 falls x=0
x*f(x–1) falls x>0

Diese Definition kann nahezu eins zu eins in eine Modula-2-Funktion übertragen werden:

```
PROCEDURE f(x : CARDINAL): CARDINAL;
  BEGIN
    IF x=0
    THEN RETURN 1
    ELSE RETURN x*f(x-1)
    END
  END f;
```

Was passiert nun beim Aufruf von f(5)?

```
f(5) =  5*f(4)              (da 5<>0)
     =  5*4*f(3)            (f(4)=4*f(3))
     =  5*4*3*f(2)          (f(3)=3*f(2))
     =  5*4*3*2*f(1)        (f(2)=2*f(1))
     =  5*4*3*2*1*f(0)      (f(1)=1*f(0))
     =  5*4*3*2*1*1         (f(0)=1)  Rekursionsabbruch!
     =  120
```

Vergleicht man diese Lösung mit den Versionen, die bei der WHILE- und FOR-Anweisung vorgestellt wurden, so erkennt man, daß die iterativen Lösungen (Iteration = Wiederholung) wesentlich schneller ausgeführt werden. Die Selbstaufrufe einer Prozedur benötigen Zeit und Speicherplatz, da jedesmal eine neue Umgebung (lokale Variable, Parameter) erzeugt werden muß. Im Fall der Fakultät ist vom Standpunkt der Programmeffizienz auf jeden Fall die Lösung mit der FOR-Schleife vorzuziehen.

Wenn man andererseits in einem Mathematikbuch die Fakultät sucht, wird ausschließlich die rekursive Definition gefunden. Wie gesehen, kann diese Formulierung problemlos übernommen und in einer adäquaten Programmstruktur dargestellt werden. Bei der Fakultät ist das Finden einer gleichwertigen iterativen Form sehr einfach. In vielen anderen Fällen ist dieser Weg nicht unmittelbar gegeben, so daß dann auf eine rekursive Programmierung nicht verzichtet werden kann.

8.5 QUICKSORT

Während beim vorhergehenden Beispiel noch eine einfache iterative Lösung gefunden werden konnte, ist das Folgende eine Paradeanwendung für die Rekursion.

8.5.1 Überlegung

Es geht dabei um das Sortieren eines Feldes (ARRAY[a..b] OF FeldElement). Der Sortieralgorithmus basiert auf dem Prinzip der Grobsortierung. Dabei wird die zu sortierende Datenmenge erst einmal in zwei Haufen geteilt, wobei alle Elemente des ersten kleiner als die des zweiten sind (Beispiel bei Karteikarten: erster Haufen die Buchstaben A bis K, zweiter Haufen I bis Z). Dann wird jeder der beiden Haufen nach demselben Verfahren (vor-)sortiert usw, bis die Haufen nur noch aus einem Element bestehen.

8.5.2 Konzept

Man kann somit folgendes Konzept für das Sortieren aufstellen:

Sortieren eines (Teil-)Feldes:
1. Wenn das Feld nur aus einem Element besteht, so ist es sortiert.

2. Ansonsten wird es so in zwei Teilfelder zerlegt, daß alle Elemente des einen kleiner als die des anderen sind. Dann wird jedes Teilfeld sortiert.

8.5.3 Ausführung

Ein Teilfeld wird durch die Angabe von Anfangs- und Endindex festgelegt.

```
PROCEDURE sortiere(VAR F : Feld; von, bis : CARDINAL);
  VAR von1, bis1, (* Indizes der Teilfelder *)
      von2, bis2 : CARDINAL;
  BEGIN
    IF bis>von  (* nur wenn das Teilfeld aus mehr als einem Element besteht *)
    THEN
      zerlege(F,von,bis,von1,bis1,von2,bis2);
      sortiere(F,von1,bis1);
      sortiere(F,von2,bis2)
    END
  END sortiere;
```

Jetzt muß die – noch nicht ausgeführte – Prozedur «zerlege» realisiert werden. Dazu benutzen wir zwei Hilfsvariable i und j. Am Anfang wird i auf von, j auf bis gesetzt. Ebenfalls am Anfang wird ein mittleres Element bestimmt. Wir wählen dazu das Element in der physikalischen Feldmitte. Dann wird, solange i kleiner als j ist, folgendes Verfahren praktiziert:

- i wird so lange erhöht, bis ein Feldelement gefunden ist, das größer als oder gleich wie das mittlere Element ist.
- j wird so lange erniedrigt, bis ein Feldelement gefunden wird, das kleiner als das mittlere Element ist.

Falls i<j ist, werden die Feldelemente mit den Indizes i und j miteinander vertauscht und schließlich noch i erhöht und j erniedrigt.

Nach dem Durchlauf gilt – davon sollten Sie sich überzeugen –, daß alle Feldelemente zwischen «von» und «j» kleiner sind als die zwischen «i» und «bis». Die Feldelemente zwischen «j» und «i» sind genauso groß wie das mittlere Element und befinden sich bereits an der richtigen Stelle. Somit ergibt sich für die zerlegten Teilfelder:

von1:=von; bis1:=j;
von2:=i; bis2:=bis;

Die entsprechende Prozedur:

```
PROCEDURE zerlege(VAR F : Feld; von,bis : CARDINAL;
                  VAR von1,bis1,von2,bis2 : CARDINAL);
  VAR i,j : CARDINAL;
      Mitte : FeldElement;
  BEGIN
    Mitte:=F[(von+bis) DIV 2]; (* die physikalische Mitte *)
    i:=von; j:=bis;
    WHILE i<j DO
      WHILE F[i]<Mitte DO INC(i) END;
      WHILE Mitte<F[j] DO DEC(j) END;
      IF i<j
      THEN vertausche(F[i],F[j]); INC(i); DEC(j)
      END (* IF *)
    END; (* WHILE *)
    von1:=von; bis1:=j;
    von2:=i; bis2:=bis
  END zerlege;
```

8.6 Ausfüllen geschlossener Flächen

Das nächste Beispiel stammt aus der Computergrafik. Dabei wird der Bildschirm als Zeichenfläche aufgefaßt, auf der jeder Punkt durch Angabe seiner Koordinaten (x,y) gesetzt (SetzePunkt(x,y)) werden kann. Die boolesche Funktion «IstPunktGesetzt(x,y)» gibt darüber Auskunft, ob der Punkt (x,y) gesetzt ist (TRUE) oder nicht (FALSE). Gesucht ist nun eine Prozedur, die eine beliebig geschlossene Figur ausmalt. Sie soll als einzige Parameter die Koordinaten eines Punktes innerhalb dieser Figur erhalten.

Ein erster Ansatz zur Lösung dieses Problems kann so formuliert werden:

Falls der Punkt nicht gesetzt ist, dann male ihn aus und führe das Verfahren für alle vier Nachbarpunkte aus.

Wenn Sie sich eine kleine Skizze machen, werden Sie schnell erkennen, daß das Verfahren wirklich zum Erfolg führt.

```
PROCEDURE MaleAus1(x,y:CARDINAL);
  BEGIN
    IF NOT IstPunktGesetzt(x,y)
    THEN
      SetzePunkt(x,y);
      MaleAus(x+1,y);
      MaleAus(x-1,y);
      MaleAus(x,y+1);
      MaleAus(x,y-1)
    END (* IF *)
  END MaleAus1;
```

Wenn Sie mit einem grafikfähigen System arbeiten, so sollten Sie diese Prozedur testen. Bei Ihren Experimenten werden Sie folgende Feststellungen machen:

1. Das Verfahren funktioniert bei kleinen Figuren, ist aber sehr langsam und macht manchmal «seltsame Pausen».
2. Bei größeren Figuren, auch wenn diese einfache geometrische Formen aufweisen, bricht das Programm mit einer bislang unbekannten Fehlermeldung ab.

Hier kommen wir zu einem echten Problem bei rekursiven Algorithmen auf Rechenanlagen. Wie schon gesagt, benötigt jeder Selbstaufruf einer Prozedur einen bestimmten Speicherplatz. Selbst wenn keine lokalen Variablen und Parameter vorliegen, muß doch jedesmal die Rücksprungadresse gespeichert werden, damit nach der Ausführung das Programm an der richtigen Stelle fortgesetzt werden kann. Aber auch der Speicherplatz des größten Computers ist begrenzt und damit auch die Anzahl der rekursiven Aufrufe. Die obige Prozedur ist extrem rekursiv, weshalb bei großen Figuren die Arbeit wegen Speichermangels einfach abgebrochen wird. Die Prozedur «bewegt sich» ausschließlich durch Selbstaufrufe erst einmal bis an den Rand einer Figur. Um später wieder an den Ausgangspunkt zurückzugelangen (von dort aus werden ja dann die Nachbarpunkte untersucht), wird genauso oft, wie der Selbstaufruf erfolgte, der Prozedurrumpf ohne sichtbare Aktion ausgeführt. Das äußert sich hier in sichtbaren Pausen, die während des Ausmalens eingelegt werden. In Informatikerkreisen wird dieses unschöne Verhalten ganz bildhaft als «Nachklappern» der Rekursion bezeichnet.

Beim Einsatz rekursiver Algorithmen ist also unbedingt darauf zu achten, daß

- ☐ die Rekursionstiefe (maximale Anzahl direkt hintereinander ausgeführter Selbstaufrufe) möglichst klein ist und
- ☐ das «Nachklappern» vermieden wird.

Ein Weg zum Erfüllen dieser Forderung liegt darin, das Problem so weit wie möglich iterativ zu lösen und nur dann auf rekursive Aufrufe zurückzugreifen, wenn es nicht mehr anders geht. Es gibt zwar einen Fundamentalsatz der Informatik, der besagt, daß jeder rekursive Algorithmus in einen iterativen überführt werden kann. In dem Augenblick, in dem das iterative Äquivalent nur dadurch realisiert werden kann, daß die Schaffung neuer Umgebungen und

die Zwischenspeicherung von Ergebnissen und noch nicht ausgeführter Teilaufgaben expliziter Teil des Algorithmus werden, ist der Einsatz der rekursiven Lösung grundsätzlich vorzuziehen. Ein Beispiel für eine solche iterative Form findet der interessierte Leser in «N. Wirth, Algorithmen und Datenstrukturen mit Modula-2, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1985».

Wir stehen hier erstmals vor einem Problem, das in der Programmierpraxis oft auftritt: Ein (an sich ideal einfacher) Algorithmus ist praktisch nicht einsetzbar. Die Prozedur «MaleAus» muß wesentlich verbessert werden. Zum Verhindern überflüssiger Aufrufe bietet sich folgendes Verfahren an:

Wenn der Punkt (x,y) nicht gesetzt ist, dann führe folgendes aus:

Gehe vom Punkt (x,y) erst so weit nach links, bis ein Randpunkt erreicht wird. Speichere den letzten Punkt vor dem Rand in (l,y). Gehe vom Punkt (x,y) so weit nach rechts, bis ein Randpunkt erreicht wird. Speichere den letzten Punkt vor dem Rand in (r,y).Ziehe eine Linie von (l,y) bis (r,y).

Führe dasselbe Verfahren für die Punkte (i,y+1) und (i,y−1) durch, wobei i alle Werte von l bis r annimmt.

Hier wird wenigstens ein Teil der rekursiven Aufrufe gespart. Trotzdem ist das Verfahren vollständig, es wird nur ein Teil der Arbeit iterativ ausgeführt. Da nur waagerechte Linien gezogen werden, bietet sich folgende Prozedur an:

```
PROCEDURE ZieheLinie(x1,y1,x2,y2:CARDINAL); (* y1 = y2 ! *)
  VAR i : CARDINAL;
  BEGIN
    FOR i:=x1 TO x2 DO SetzePunkt(i,y1) END
  END ZieheLinie;
```

Die Übersetzung der Vorschrift in eine Prozedur ist nicht schwer:

```
PROCEDURE MaleAus2(x,y : CARDINAL);
  VAR l,r,i : CARDINAL;
  BEGIN
    IFIstPunktGesetzt(x,y) THEN RETURN END;
    l:=x; WHILE NOT IstPunktGesetzt(l,y) DO DEC(l) END;
    r:=x; WHILE NOT IstPunktGesetzt(r,y) DO INC(l) END;
    INC(l); DEC(r); (* Warum ? *)
    ZieheLinie(l,y,r,y);
    FOR i:=l TO r DO
      MaleAus2(i,y+1);
      MaleAus2(i,y-1)
    END (* FOR *)
  END MaleAus2;
```

Mit dieser Prozedur können schon etwas größere Figuren ausgemalt werden. Dennoch werden übermäßig viele unnötige Tests durchgeführt. Bei einer einfachen Figur wie einem Rechteck beispielsweise führt bereits der erste Aufruf von «MaleAus2(i,y+1)» dazu, daß die Linie (l,y+1) bis (r,y+1) gezeichnet wird. Trotzdem wird die FOR-Schleife stur fortgesetzt und laufend «MaleAus2» aufgerufen, die jedesmal prompt über RETURN verlassen wird, da diese Punkte schon gesetzt sind. Ausweg bietet das Ersetzen der FOR-Anweisung durch zwei «intelligentere» WHILE-Anweisungen. Dazu muß jedoch nach außen bekanntgemacht werden, welche Linie bei einem Aufruf gezogen wurde. Aus diesem Grund bauen wir in «MaleAus3» eine verschachtelte Prozedur ein, die die bisherige Arbeit von «MaleAus» übernimmt, die Randpunkte der Linie jedoch als Referenzparameter nach außen gibt.

```
PROCEDURE MaleAus3(x,y : CARDINAL);
  VAR MaxLinks, MaxRechts : CARDINAL;

  PROCEDURE Male(x,y : CARDINAL; VAR l,r : CARDINAL);
    VAR links ,rechts : CARDINAL;
    BEGIN
      IF IstPunktGesetzt(x,y) THEN RETURN END;
      l:=x; WHILE NOT IstPunktGesetzt(l,y) DO DEC(l) END;
      r:=x; WHILE NOT IstPunktGesetzt(r,y) DO INC(l) END;
      INC(l); DEC(r);
      ZieheLinie(l,y,r,y);
      links:=l;
      WHILE links<=r DO
        Male(links,y+1,rechts,links);
        INC(links)
    END; (* WHILE *)
    links:=l;
    WHILE links<=r DO
      Male(links,y-1,rechts,links);
      INC(links)
    END (* WHILE *)
  END Male;

BEGIN (* MaleAus3 *)
  Male(x,y,MaxLinks,MaxRechts)
END MaleAus3;
```

Dieser Algorithmus arbeitet schon wesentlich besser als die vorhergehenden. Aber immer noch bleibt das Problem, daß zu viele überflüssige Aufrufe bei der Bearbeitung der senkrechten Richtung ausgeführt werden. Folglich muß auch hier eine iterative Lösung gefunden werden. Der folgende Algorithmus wurde, basierend auf den hier vorgestellten Lösungen, von meinem Kollegen Bernd Edlinger ausgearbeitet.

```
PROCEDURE MaleAus4(x,y : CARDINAL);
  VAR Anfang, Ende : CARDINAL;

  PROCEDURE LinksAussen(x,y : CARDINAL) : CARDINAL;
    BEGIN
      WHILE NOT IstPunktGesetzt(x,y) DO DEC(x) END;
      RETURN x+1
    END LinksAussen;

  PROCEDURE RechtsAussen(x,y : CARDINAL) : CARDINAL;
    BEGIN
      WHILE NOT IstPunktGesetzt(x,y) DO INC(x) END;
      RETURN x-1
    END RechtsAussen;

  PROCEDURE Male(a,e,y0,d0 : CARDINAL);
    VAR xl, xr, y, d, xlneu, xrneu : CARDINAL;
        fertig : BOOLEAN;

    PROCEDURE pruefe(l,r,y1,d1 : CARDINAL);
      VAR Test : BOOLEAN;
          x : CARDINAL;
      BEGIN
        Test:=TRUE;
        FOR x:=l TO r DO
          IF IstPunktGesetzt(x,y1)
          THEN IF NOT Test
               THEN e:=x-1; Test:=TRUE
               END
          ELSIF Test
          THEN
            IF NOT fertig THEN Male(a,e,y0,d0) END;
            a:=x; y0:=y1; d0:=d1;
            fertig:=FALSE; test:=FALSE
          END; (* IF *)
        END; (* FOR *)
        IF NOT Test THEN e:=r
      END pruefe;

    BEGIN (* Male *)
      REPEAT
        xl:=b; xr:=e; y:=y0; d:=d0;
        fertig:=TRUE;
        xlneu:=LinksAussen(xl-1,y);
        xrneu:=RechtsAussen(xr+1,y);
        ZieheLinie(xlneu,y,xrneu,y);
        pruefe(xlneu,xl-2,y-d,-d);
        pruefe(xr+2,xrneu,y-d,-d);
        pruefe(xlneu,xrneu,y+d,d)
      UNTIL fertig
    END Male;

  BEGIN (* MaleAus4 *)
    Anfang:=LinksAussen(x,y); Ende:=RechtsAussen(x,y);
    IF Anfang<=Ende
    THEN
      Male(Anfang,Ende,y,1);
      Male(Anfang,Ende,y,-1)
    END
  END MaleAus4;
```

8.7 Ausdrücke

Wie bereits angesprochen, ist in Modula-2 auch der Begriff «Anweisung» rekursiv definiert. Auch bei der exakten Festlegung des Terminus «Ausdruck» kommt die Rekursion ins Spiel. Wir wollen uns einmal die Syntax von einfachen CARDINAL-Ausdrücken ansehen:

```
CARDINAL-Ausdruck::=Term {AdditionsOperator Term}.
Term::=Faktor {MultiplikationsOperator Faktor}.
Faktor::=CARDINAL-Zahl | "(" CARDINAL-Ausdruck ")" |
        FunktionsName [AktuelleParameter].
```

Wir haben es hier mit einer indirekten Rekursion zu tun. «CARDINAL-Ausdruck» wird mit «Term» definiert, «Term» mit «Faktor» und «Faktor» schließlich wieder mit «CARDINAL-Ausdruck». Unser nächstes Beispiel soll die Syntax eines eingegebenen CARDINAL-Ausdrucks überprüfen und gegebenenfalls eine passende Fehlermeldung bringen.

Interessant an der Lösung ist die unmittelbare Umsetzung der Definition in entsprechende Modula-2-Prozeduren. Bei unserem Beispiel lassen wir die Funktionsnamen weg.

```
Faktor::=CARDINAL-Zahl | "(" CARDINAL-Ausdruck ")".
CARDINAL-Zahl::=Ziffer{Ziffer}.
```

Auch die Definition von CARDINAL-Zahlen ist nicht vollständig, auf Hexadezimal- und Oktalkonstante wird hier verzichtet.

Der zu untersuchende Ausdruck wird zunächst in eine String-Variable (ARRAY[0..80] OF CHAR) eingelesen. Die CARDINAL-Variable «Position» zeigt immer auf das gerade analysierte Zeichen. Es wird davon ausgegangen, daß am Ende des Ausdrucks mindestens ein 0C-Zeichen enthalten ist. Eventuell vorkommende Leerzeichen müssen überlesen werden. Das leistet die folgende Prozedur «LeerzeichenUeberlesen»:

```
VAR Fehler : BOOLEAN;
    Position : CARDINAL;
    Eingabe : ARRAY[0..80] OF CHAR;

    ...
```

```
PROCEDURE LeerzeichenUeberlesen;
  BEGIN
    WHILE Eingabe[Position]=" " DO INC(Position) END
  END LeerzeichenUeberlesen;
```

Als nächstes schreiben wir eine Funktion, die aus der Stringvariablen eine CARDINAL-Zahl überliest.

```
PROCEDURE CardinalZahl;
  BEGIN
    WHILE (Eingabe[Position]>="0") AND (Eingabe[Position]<="9") DO
      INC(Position)
    END
  END CardinalZahl;
```

Jetzt wird die Prozedur «Faktor» realisiert. Betrachten Sie bitte die Syntax von «Faktor». Falls das nächste Zeichen eine Ziffer ist, besteht der Faktor ausschließlich aus einer CARDINAL-Zahl. Ist das nächste Zeichen hingegen die öffnende runde Klammer, so muß ein CARDINAL-Ausdruck vorliegen. Auf diesen muß die schließende runde Klammer folgen. Ist das nicht der Fall, so liegt ein Klammerfehler vor. Falls weder eine CARDINAL-Zahl noch eine öffnende Klammer kommt, liegt ein Syntaxfehler vor.

```
PROCEDURE Faktor;
  BEGIN
    LeerzeichenUeberlesen;
    IF (Eingabe[Position]>="0") AND (Eingabe[Position]<="9")
    THEN CardinalZahl
    ELSIF Eingabe[Position]="("
    THEN
      INC(Position);
      CardinalAusdruck;
      IF Fehler THEN RETURN END;
      LeerzeichenUeberlesen;
      IF Eingabe[Position]<>")"
      THEN
        Fehlermeldung('")" fehlt!',Position);
        Fehler:=TRUE
      END
    ELSE
      Fehlermeldung("Illegale Zeichen",Position);
      Fehler:=TRUE
    END (* IF *)
  END Faktor;
```

Jetzt kommt der Begriff «Term» an die Reihe. Zur Vereinfachung der Aufgabe wollen wir als Multiplikations-Operatoren hier * und / zulassen. Die Analyse von Wortsymbolen wie «DIV» und «MOD» ist zwar nicht besonders kompliziert, lenkt aber vom wesentlichen Aspekt der rekursiven Programmierung zu stark ab.

```
PROCEDURE Term;
  BEGIN
    Faktor; LeerzeichenUeberlesen;
    WHILE NOT Fehler AND
          ((Eingabe[Position]="*") OR (Eingabe[Position]="/")) DO
      INC(Position); Faktor; LeerzeichenUeberlesen
    END
  END Term;
```

Bleibt schließlich noch der Begriff «CARDINAL-Ausdruck». Die Additions-Operatoren sind + und –.

```
PROCEDURE CardinalAusdruck;
  BEGIN
    Term; LeerzeichenUeberlesen;
    WHILE NOT Fehler AND
          ((Eingabe[Position]="+") OR (Eingabe[Position]="-")) DO
      INC(Position); Term; LeerzeichenUeberlesen
    END
  END CardinalAusdruck;
```

Bleibt zum Schluß nur noch die Ausführung der Prozedur «Fehlermeldung». Die soll die Fehlerstelle genau anzeigen. Dazu postieren wir unterhalb der Eingabezeile einen Pfeil ^ auf das Zeichen, auf das der übergebene Wert zeigt:

```
PROCEDURE Fehlermeldung(Meldung : ARRAY OF CHAR; ZeichenNr : CARDINAL);
  VAR i : CARDINAL;
  BEGIN
    FOR i:=1 TO ZeichenNr-1 DO Write(" ") END;
    Write("^"); WriteLn;
    WriteString(Meldung); WriteLn
  END Fehlermeldung;
```

Ein Programm, das die syntaktische Analyse eines (Programm-) Textes vornimmt, nennt man normalerweise «Parser». Aus diesem Grund heißt unser Modul «AusdruckParser».

```
MODULE AusdruckParser;
  FROM InOut IMPORT Write, WriteString, WriteLn, ReadString;
  (* Deklarationsteil wie oben inklusive aller Prozeduren *)

  BEGIN
    WriteString("Syntaktische Analyse von CARDINAL-Ausdrücken"); WriteLn;
    WriteString("-----------------------------------------"); WriteLn;
    WriteLn;
    WriteString("Zugelassene Rechenzeichen: + - * / ( )"); WriteLn;
    WriteLn;
    WriteString("Ihr Ausdruck: "); WriteLn;
    ReadString(Eingabe); WriteLn;
    Position:=0; Fehler:=FALSE;
    CardinalAusdruck
  END AusdruckParser.
```

Probelauf:

```
Syntaktische Analyse von CARDINAL-Ausdrücken

Zugelassene Rechenzeichen: + - * / ( )

Ihr Ausdruck:
(1+14*(48-19*3
               ^
")" fehlt!
```

8.8 Indirekte Rekursion in verschachtelten Blöcken

Im Ausdruckparser wird vom Hauptprogramm nur die Prozedur «CardinalAusdruck» aufgerufen, «Faktor», «Term» und «CardinalZahl» rufen sich nur gegenseitig bzw. werden von «CardinalAusdruck» gerufen. Aus diesem Grund bietet es sich an, diese Prozeduren lokal zu «CardinalAusdruck» zu deklarieren und somit aus der indirekten Rekursion eine quasi-direkte zu machen. Es ist sicher ein sinnvoller Gedanke, so lokal wie nur irgendwie möglich zu programmieren.

```
PROCEDURE CardinalAusdruck;

  PROCEDURE LeerzeichenUeberlesen;
    BEGIN
      WHILE Eingabe[Position]=" " DO INC(Position) END
    END LeerzeichenUeberlesen;

  PROCEDURE Term; (* wird nur von 'CardinalAusdruck' gerufen *)

    PROCEDURE Faktor; (* wird nur von 'Term' gerufen *)

      PROCEDURE CardinalZahl; (* wird nur von Faktor gerufen *)
        BEGIN
          WHILE (Eingabe[Position]>="0") AND (Eingabe[Position]<="9") DO
            INC(Position)
          END
        END CardinalZahl;

      BEGIN (* Faktor *)
        LeerzeichenUeberlesen;
        IF (Eingabe[Position]>="0") AND (Eingabe[Position]<="9")
        THEN CardinalZahl
        ELSIF Eingabe[Position]="("
        THEN
          INC(Position);
          CardinalAusdruck;
```

```
        IF Fehler THEN RETURN END;
        LeerzeichenUeberlesen;
        IF Eingabe[Position]<>")"
        THEN
          Fehlermeldung('")" fehlt!',Position);
          Fehler:=TRUE
        END
      ELSE
        Fehlermeldung("Illegale Zeichen",Position);
        Fehler:=TRUE
      END (* IF *)
    END Faktor;

  BEGIN (* Term *)
    Faktor; LeerzeichenUeberlesen;
    WHILE NOT Fehler AND
          ((Eingabe[Position]="*") OR (Eingabe[Position]="/")) DO
      INC(Position); Faktor; LeerzeichenUeberlesen
    END
  END Term;

BEGIN (* CardinalAusdruck *)
  Term; LeerzeichenUeberlesen;
  WHILE NOT Fehler AND
        ((Eingabe[Position]="+") OR (Eingabe[Position]="-")) DO
    INC(Position); Term; LeerzeichenUeberlesen
  END
END CardinalAusdruck;
```

Der rekursive Aufruf steckt in der lokalen Prozedur «Faktor», von der aus auf die globale Prozedur «CardinalAusdruck» zugegriffen wird.

Aufgaben:

1. Definieren Sie den Begriff «Vorfahre» mit den Begriffen «Vater» und «Mutter».
2. Schreiben Sie die Prozedur «vertausche» (Quicksort).
3. Fügen Sie die einzelnen Teile so zusammen, daß die Prozedur «sortiere» vollständig ist.
4. Schreiben Sie ein Testprogramm, das ein Feld von 1000 Integer-Zahlen zufällig erzeugt, mit «sortiere» bearbeitet und das Ergebnis ausgibt.
5. Machen Sie mit dem Programm aus Aufgabe 3 eine Testreihe, bei der Sie die Feldgröße variieren. Legen Sie eine Tabelle mit Feldgröße und Sortierzeiten an.
6. Ändern Sie das Programm so, daß STRINGs sortiert werden.
7. Welche Änderungen sind an «AusdruckParser» vorzunehmen, daß der Wert eines Ausdrucks berechnet wird?

9 Dynamische Datenstrukturen

Alle bisher bekannten Datentypen sind statischer Natur. Das heißt, daß eine Variable (eines solchen Typs) genau so lange existiert wie die Umgebung, in der sie deklariert wurde. Die globalen Variablen eines Hauptprogramms z. B. sind während der gesamten Laufzeit des Programms vorhanden.

Es gibt jedoch Aufgaben, bei denen manche Variable nur unter bestimmten Umständen benötigt werden. Ein häufig auftretendes Problem besteht auch darin, daß die Anzahl der benötigten Variablen nicht bekannt ist. Die Lösung mit Feldern ist in diesem Fall immer mit einem Unsicherheitsfaktor verknüpft.

Wir wollen diese Problematik an einem Beispiel darstellen. Es soll ein Programm geschrieben werden, das die verschiedenen Wörter eines Textes zählt. Die Größe der zu untersuchenden Texte beträgt jeweils ca. 100 KByte. Die Datenstruktur würde mit den vorhandenen Mitteln etwa so aussehen:

```
CONST MaxWortLaenge = ??
      MaxWortZahl = ??

TYPE  WortVorkommen = RECORD
                        Wort : ARRAY[0..MaxWortLaenge] OF CHAR;
                        Anzahl : integer
                      END;

VAR   WortListe : ARRAY[1..MaxWortZahl] OF WortVorkommen;
      GesamtZahl : 0..MaxWortZahl;
```

Offen bleiben dabei noch die Konstanten «MaxWortLaenge» und «MaxWortZahl». Um auch längere Wörter speichern zu können, muß hier eine Zahl ab 30 eingesetzt werden, obwohl – aller Voraussicht nach – die durchschnittliche Wortlänge weit unter 10 bleibt. Es ist also schon an dieser Stelle offenkundig, daß die Variable «WortListe» zu zwei Dritteln leer bleibt. Andererseits besteht weiterhin die Gefahr, daß der Text Wörter mit größerer Länge enthält, die dann nicht mehr richtig verarbeitet werden.

Noch unsicherer ist die Bestimmung von «MaxWortZahl». Selbst wenn hier die Zahl 10000 eingesetzt wird (mittlere Wortlänge = 5, mittlere Häufigkeit = 2), ist nicht sichergestellt, daß wirklich alle vorkommenden Wörter gespeichert werden können. Zudem belegt das resultierende Feld einen Speicherbereich größer als 300 KByte (MaxWortLaenge * MaxWortZahl), wovon mit Sicherheit mehr als 65 % überflüssig ist.

Eine weitere Schwierigkeit tritt auf, wenn die entsprechenden Algorithmen formuliert werden sollen. Bei jedem eingelesenen Wort muß geprüft werden, ob es bereits in der Liste vorhanden ist. Wenn ja, so muß der entsprechende Zähler weitergezählt werden, ansonsten ist es neu in die Liste einzufügen. Die schnelle binäre Suche funktioniert bekanntlich nur in einem geordneten Feld. Da es wenig Sinn macht, das Feld nach jedem Einfügen neu zu sortieren oder zu organisieren (die Einfügestelle bestimmen und den Rest des Feldes verschieben), bleibt nur die zeitaufwendige lineare Suche.

Die Suche nach einem Ausweg aus diesem Dilemma führt zu den dynamischen Datenstrukturen. Denn alle diese Probleme können gelöst werden, wenn

1. die Möglichkeit besteht, Variable bei Bedarf und in gewünschter Anzahl und Größe während des Programmablaufs zu erzeugen, und
2. diese Variablen so strukturiert werden können, daß auf sie (je nach Problem) in möglichst effizienter Weise zugegriffen werden kann.

9.1 Der Pointer-Typ

Nun würde das gesamte Konzept einer Programmiersprache wie Modula-2 ad absurdum geführt, könnte man auf nicht deklarierte Variable nach Bedarf zugreifen und mit diesen arbeiten. Vielmehr muß hier der Wunsch angemeldet werden, daß eine Variable eines bestimmten Typs benötigt werden könnte. Diese Option auf eine Variable wird durch den Ausdruck «POINTER TO», der dem gewünschten Typ vorangestellt wird, ausgedrückt.

Zeigertyp::="POINTER" "TO" Typ.

Beispiel:

```
TYPE Wort =ARRAY[0..30] OF CHAR;
VAR  EventuellGebrauchtesWort : POINTER TO Wort;
```

9.1.1 Erzeugen von dynamischen Variablen

Die Variable «EventuellGebrauchtesWort» ist nicht vom Typ «Wort»! Sie ist vielmehr ein Hinweis (Zeiger, Pointer) auf eine (noch nicht existierende) Variable dieses Typs. Wird nun während des Programmablaufs eine Variable vom Typ «Wort» benötigt, so muß zunächst der entsprechende Speicherplatz bereitgestellt werden. Dafür gibt es «ALLOCATE» aus dem Modul «Storage» (in manchen Systemen auch «STORAGE»).

ALLOCATE(ZeigerVariable, AnzahlSpeicherworte)

Der Aufruf dieser Prozedur bewirkt, daß für die Variable, auf die die Zeigervariable hinweist (zeigt), ein Speicherbereich von Anzahl-Speicherworte zur Verfügung gestellt wird. Der benötigte Speicherbereich kann mit der Standardfunktion «TSIZE» aus dem Modul «Storage» bestimmt werden.

Beispiel:

ALLOCATE(EventuellGebrauchtesWort,TSIZE(Wort))

Da normalerweise der benötigte Speicherplatz immer gleich dem einer statisch vereinbarten Variablen (VAR NeuesWort : Wort) ist, gibt es in manchen Modula-2-Systemen für die Konstruktion

```
TYPE  IrgendeinTyp = ...
VAR   ZeigerVar : POINTER TO IrgendeinTyp;

 ALLOCATE(ZeigerVar,TSIZE(IrgendeinTyp))
```

die Standardprozedur "NEW":

```
... NEW(ZeigerVar)
```

Hinweis: Da der Compiler zur Übersetzung von «NEW» auf die Prozedur «ALLOCATE» zurückgreift, muß diese in jedem Fall importiert werden. Da insbesondere bei neueren Systemen die

Verfügbarkeit von «NEW» nicht sichergestellt ist, wird in der Folge ausschließlich «ALLOCATE» verwendet.

Nach «ALLOCATE» (oder «NEW») existiert somit eine neue Variable. Diese hat keinen eigenen Namen (deshalb auch oft «anonyme Variable»). Der Zugriff erfolgt über die zugehörige Zeigervariable, indem an diese ein Hochpfeil angehängt wird. Der Ausdruck «ZeigerVar^» ist am besten so zu lesen: «Diejenige Variable, auf die ZeigerVar zeigt». In der Folge entspricht diese Variable genau einer normal (statisch) deklarierten. Da sie erst während der Laufzeit durch Ausführung der Prozedur «ALLOCATE» entstanden ist, nennt man sie «dynamisch erzeugt».

Beispiel:

```
MODULE ZeigerDemo;

  FROM Storage IMPORT ALLOCATE;
  FROM SYSTEM IMPORT TSIZE;
  FROM InOut IMPORT WriteString, ReadString, WriteLn, Read;

  TYPE Wort = ARRAY[0..30] OF CHAR;
  VAR  EventuellGebrauchtesWort : POINTER TO Wort;
       Antwort : CHAR;

  BEGIN
    WriteString("Wollen Sie ein Wort eingeben (J/N) ?");
    REPEAT
      Read(Antwort); Antwort:=CAP(Antwort)
    UNTIL (Antwort='J') OR (Antwort='N');
    WriteLn;
    IF Antwort='J'
    THEN
      ALLOCATE(EventuellGebrauchtesWort,TSIZE(Wort));
      Write('Ihr Wort: ');
      ReadString(EventuellGebrauchtesWort^); WriteLn;
      WriteString('Ihr Wort war: ');
      WriteString(EventuellGebrauchtesWort^);
      WriteLn
    END (* IF *)
  END.
```

9.1.2 Löschen von dynamischen Variablen

Wenn eine derart dynamisch erzeugte Variable nicht mehr benötigt wird, kann der belegte Speicherplatz wieder freigegeben werden. Dafür sind die Prozeduren «DEALLOCATE» und «DISPOSE» (als Gegenstück zu «NEW») zuständig. «DEALLOCATE» muß aus dem Modul «Storage» importiert werden, auch wenn «DISPOSE» eingesetzt wird.

DEALLOCATE(ZeigerVariable,AnzahlSpeicherworte)

ist das Gegenstück zu «ALLOCATE». Wichtig bei einem Einsatz ist, daß «AnzahlSpeicherworte» genauso groß sein muß wie bei «ALLOCATE». Handelt es sich bei «AnzahlBytes» um dieselbe Größe wie bei einer statischen Variablen, so kann hier als Pendant zu «NEW» die Prozedur «DISPOSE(ZeigerVariable)» eingesetzt werden. Dabei gelten dieselben Einschränkungen wie bei «NEW».

9.1.3 Gefahren bei der Arbeit mit dynamischen Variablen

Der Wert von Zeigervariablen sind Speicheradressen. «ALLOCATE» (oder «NEW») weist einer Zeigervariablen eine Adresse zu, an der sich ein bislang unbenutzter Speicherbereich befindet. Zusätzlich wird der benötigte Speicherbereich intern als belegt gekennzeichnet. Wird nun eine Zeigervariable ohne vorhergehendes «ALLOCATE» (oder «NEW») benutzt, so kann das mitunter zu einem Absturz des gesamten Systems führen. Da in Modula-2 Variable normalerweise nicht initialisiert werden, hat eine Zeigervariable erst einmal einen zufälligen Wert, sie zeigt also auf irgendeinen Speicherbereich. Eine Zuweisung an die Variable, auf die diese Zeigervariable zeigt, hat somit zur Folge, daß in irgendwelche Speicherzellen Werte eingetragen werden. Da es sich dabei um wichtige, vom Betriebssystem benötigte Speicherbereiche handeln kann, ist ein mögliches Chaos vorgezeichnet.

Auch «DEALLOCATE» oder «DISPOSE» dürfen nicht verwendet werden, bevor eine entsprechende Variable geschaffen wurde. Die Folgen können ähnlich den oben beschriebenen sein.

9.2 Dynamische Strings in Modula-2

Ein erster Einsatz für Zeigervariable ist die Bereitstellung von dynamischen Strings, die

- Zeichenketten beliebiger Länge beinhalten können und
- nur den benötigten Speicherbereich belegen.

In Modula-2 werden Zeichenketten üblicherweise als «ARRAY-[0..MaxZeichen] OF CHAR» dargestellt. Bei Zuweisungen werden die überflüssigen Zeichen mit 0C aufgefüllt. Es ist jedoch ganz

offensichtlich, daß zur Kennzeichnung des Endes einer Zeichenkette nur ein (!) 0C nötig ist.

Die Idee der dynamischen Strings beruht nun darauf, daß zur Eingabe eine statische (Puffer-)Variable benutzt und zur Speicherung nur der unbedingt benötigte Speicherplatz belegt wird. Da dynamische Strings sehr häufig benötigt werden, schreiben wir ein neues Bibliotheksmodul mit dem Namen «DynStr».

Allerdings funktioniert das Verfahren nur, wenn bekannt ist, wie der jeweilige Compiler Variable vom Typ String behandelt. Denn in vielen Fällen wird ein «ARRAY[0...] OF CHAR» byteweise abgelegt. Zur Bestimmung der benötigten Anzahl von Speicherworten wird im Implementationsteil die Variable «Packung» entsprechend gesetzt. Wir erinnern uns, daß der Anweisungsteil eines Implementations-Moduls vor dem ersten Zugriff auf einen importierten Bezeichner ausgeführt wird.

Achtung: In einigen älteren Systemen muß vor der Konstantenvereinbarung eine Exportliste mit allen exportierten Bezeichnern stehen, also

```
  EXPORT QUALIFIED MaxStringLaenge, StatString, DynString, ...

DEFINITION MODULE DynStr;

  (* Liefert alle wichtigen Prozeduren zur Arbeit mit dynamischen
     Strings. Dynamische Strings eignen sich zur effizienten Speicherung
     von Zeichenketten. Die eigentliche Textverarbeitung (Einfügen,
     Verkürzen etc.) sollte hingegen nur mit statischen Strings gemacht
     werden *)

  CONST MaxStringLaenge = 80;

  TYPE  StatString = ARRAY[0..MaxStringLaenge] OF CHAR;
        DynString = POINTER TO StatString;

  PROCEDURE ReadDynString(VAR S : DynString);
   (* Liest einen dynamischen String mit Read aus InOut.
      Achtung: Falls S bereits einen Wert hat, ist dieser verloren! *)

  PROCEDURE WriteDynString(S : DynString);
   (* Gibt einen dynamischen String mit Write aus InOut aus. Dadurch
      bleiben Ausgabeumleitungen wirksam *)

  PROCEDURE MakeDynString(StatString : ARRAY OF CHAR; VAR S : DynString);
   (* Fertigt die dynamische Kopie eines statischen Strings an.
      Achtung: Falls S bereits einen Wert hat, ist dieser verloren! *)

  PROCEDURE MakeStatString(VAR StatString : ARRAY OF CHAR; S : DynString);
   (* Kopiert einen dynamischen String in einen statischen. Falls der
      dynamische String zu lang ist, wird der Rest abgeschnitten *)
```

```
  PROCEDURE ForgetDynString(VAR S : DynString);
   (* Gibt den Speicherplatz wieder frei. Nach dem Aufruf dieser
      Prozedur ist der Inhalt von S verloren *)

  PROCEDURE DynStringLength(S : DynString):CARDINAL;
   (* Liefert die Länge eines dynamischen Strings *)

  PROCEDURE DynStringLess(S1, S2 : DynString):BOOLEAN;
   (* Vergleicht zwei dynamische Strings. Wird TRUE, wenn der erste
      String in alphabetischer Reihenfolge vor dem zweiten kommt,
      sonst FALSE *)

END DynStr.
```

Hier folgt nun der entsprechende Implementationsteil. Bitte beachten Sie, daß die Konstante «EOL» bei manchen Systemen im Modul «ASCII» versteckt ist.

```
IMPLEMENTATION MODULE DynStr;

  FROM STORAGE IMPORT ALLOCATE, DEALLOCATE;
  FROM InOut IMPORT Read, Write, EOL;

  TYPE TestString = ARRAY[0..15] OF CHAR;
  VAR Packung : CARDINAL;

  PROCEDURE length(S : ARRAY OF CHAR) : CARDINAL;
    VAR i : CARDINAL;
    BEGIN
      i:=0;
      WHILE (i<=HIGH(S)) AND (S[i]>0C) DO INC(i) END;
      RETURN i
    END length;

  PROCEDURE MakeDynString(StatString : ARRAY OF CHAR; VAR S : DynString);
    VAR BenoetigteWorte, AnzahlBytes, i : CARDINAL;
    BEGIN
      AnzahlBytes:=length(StatString)+1;
      BenoetigteWorte:=(AnzahlBytes+Packung-1) DIV Packung;
      ALLOCATE(S,BenoetigteWorte);
      FOR i:=0 TO AnzahlBytes-1 DO S^[i]:=StatString[i] END;
      S^[AnzahlBytes]:=0C
    END MakeDynString;

  PROCEDURE MakeStatString(VAR StatString : ARRAY OF CHAR; S : DynString);
    VAR i : CARDINAL;
    BEGIN
      FOR i:=0 TO length(S^) DO
        IF i>HIGH(StatString)
        THEN RETURN
        ELSE StatString[i]:=S^[i]
        END (* IF *)
      END (* FOR *)
    END MakeStatString;

  PROCEDURE ReadDynString(VAR S : DynString);
    VAR Puffer : StatString;
        i : CARDINAL;
        Zeichen : CHAR;
```

```
  BEGIN
    i:=0;
    REPEAT
      Read(Zeichen); Puffer[i]:=Zeichen; INC(i)
    UNTIL (i>MaxStringLaenge) OR (Zeichen=EOL);
    IF Zeichen=EOL THEN Puffer[i-1]:=0C;
    MakeDynString(Puffer,S)
  END ReadDynString;

PROCEDURE WriteDynString(S : DynString);
  VAR i : CARDINAL;
  BEGIN
    i:=0; WHILE S^[i]<>0C DO Write(S^[i]); INC(i) END
  END WriteDynString;

PROCEDURE ForgetDynString(VAR S : DynString);
  VAR BenoetigteWorte, AnzahlBytes, i : CARDINAL;
  BEGIN
    AnzahlBytes:=length(S^)+1;
    BenoetigteWorte:=(AnzahlBytes+Packung-1) DIV Packung;
    DEALLOCATE(S,BenoetigteWorte)
  END ForgetDynString;

PROCEDURE DynStringLength(S : DynString):CARDINAL;
  BEGIN
    RETURN length(S^)
  END DynStringLength;

PROCEDURE DynStringLess(S1, S2 : DynString) : BOOLEAN;
  VAR i : CARDINAL;
  BEGIN
    i:=0;
    LOOP
      IF (S1^[i]=S2^[i]) AND (S1^[i]<>0C)
      THEN INC(i)
      ELSE RETURN S1^[i]<S2^[i]
      END (* IF *)
    END (* LOOP *)
  END DynStringLess;

BEGIN
  Packung:=16 DIV TSIZE(TestString)
END DynStr.
```

Als Beispiel folgt nun ein kleines Programm, das sämtliche in einem Text vorkommenden Wörter in einem Feld von dynamischen Strings ablegt und die Vorkommen zählt. Das Programm ist aufgrund der ARRAY-Struktur relativ langsam.

In der Prozedur «Einfuegen» wird zunächst ein temporärer dynamischer String (temp) aus dem übergebenen Wort erzeugt. Dann wird die gesamte vorhandene Wortliste durchsucht, ob dieses Wort bereits gelesen wurde. Die Gleichheit wird auf «DynStringLess» zurückgeführt nach der Regel:

Zwei Wörter sind gleich, wenn keines der beiden vor dem anderen kommt.

Somit wird der Ausdruck

DynStringLess(L[i].Wort,temp) OR DynStringLess(temp,L[i].Wort)

TRUE, wenn die beiden Strings ungleich sind,
FALSE, wenn sie in allen Zeichen übereinstimmen.

Wird das Wort in der Liste gefunden, so wird die Anzahl der Vorkommen erhöht und die temporäre Variable gelöscht (ForgetDynString). Andernfalls wird die temporäre Variable am Ende der Liste angehängt und die Listengröße (A) erhöht.

Bei der Prozedur «Sortiere» handelt es sich um den rekursiven Quicksort-Algorithmus in etwas komprimierter Form. Man sieht, daß auch die Zuweisung mit dynamischen Strings problemlos funktioniert, solange man sich auf Operationen wie «Vertauschen» beschränkt.

Das Einlesen der Wörter übernimmt unser bekannter Automat.

alter Zustand	gelesenes Zeichen	Aktion	neuer Zustand
ZeichenLesen	kein Buchstabe	–	–
ZeichenLesen	Buchstabe	Puffervariable initialisieren	WortLesen
WortLesen	Buchstabe	Zeichen anhängen	–
WortLesen	kein Buchstabe	Puffer einfügen	ZeichenLesen

Für die Ausgabe wurde keine eigene Prozedur geschrieben, da sie in einer einzigen FOR-Anweisung realisiert werden kann.

```
MODULE WortHaeufigkeiten;

  FROM DynStr IMPORT StatString, DynString, MakeDynString, MakeStatString,
                     ForgetDynString, DynStringLess, WriteDynString,
                     MaxStringLaenge;

  FROM InOut IMPORT Read, OpenInput, CloseInput, Done, WriteCard,
                    WriteString, WriteLn;

  CONST MaxWortZahl = 1000;
```

```
TYPE WortVorkommen = RECORD
                       Wort : DynString;
                       Anzahl : CARDINAL
                     END;
     WortListe = ARRAY[1..MaxWortZahl] OF WortVorkommen;

VAR Woerter : WortListe;
    Gesamt, i : CARDINAL;
    Puffer : StatString;
    ZeichenZahl : [0..MaxStringLaenge];
    Zeichen : CHAR;
    Zustand : (ZeichenLesen, WortLesen);

PROCEDURE Einfuegen(S : StatString; VAR A : CARDINAL; VAR L : WortListe);
  VAR i : CARDINAL;
      temp : DynString;
  BEGIN
    MakeDynString(S,temp);
    i:=1;
    WHILE (i<=A) AND (DynStringLess(L[i].Wort,temp) OR
          DynStringLess(temp,L[i].Wort)) DO INC(i) END;
    IF i<=A
    THEN INC(L[i].Anzahl); ForgetDynString(temp)
    ELSIF A<MaxWortZahl
    THEN INC(A); L[A].Wort:=temp; L[A].Anzahl:=1
    ELSE WriteString("Feld ist zu klein!"); WriteLn; HALT
    END
  END Einfuegen;

PROCEDURE Sortiere(VAR L : WortListe; von, bis : CARDINAL);
  VAR i,j : CARDINAL;
      temp, test : DynString;
  BEGIN
    i:=von; j:=bis; test:=L[(i+j) DIV 2)];
    WHILE i<j DO
    WHILE DynStringLess(L[i],test) DO INC(i);
    WHILE DynStringLess(test,L[j]) DO DEC(j);
    IF i<=j
    THEN temp:=L[i]; L[i]:=L[j]; L[j]:=temp; INC(i); DEC(j)
    END (* IF *)
  END; (* WHILE *)
  IF von<j THEN Sortiere(L,von,j) END;
  IF i<bis THEN Sortiere(L,i,bis) END
END Sortiere;

BEGIN (* WortZahl *)
  WriteString("Worthäufigkeiten in einem Text"); WriteLn;
  WriteString("------------------------------"); WriteLn;
  WriteLn;
  OpenInput(""); IF NOT Done THEN WriteString("Keine Datei!"); HALT END;
  Gesamt:=0; Zustand:=ZeichenLesen;
  Read(Zeichen);
  WHILE Done DO
    CASE Zustand OF
      ZeichenLesen : IF (CAP(Zeichen)>='A') AND (CAP(Zeichen)<='Z')
                     THEN
                       ZeichenZahl:=0;
                       Puffer[ZeichenZahl]:=Zeichen;
                       Zustand:=WortLesen
                     END
```

```
    | WortLesen     : IF (CAP(Zeichen)>='A') AND (CAP(Zeichen)<='Z')
                    THEN
                      IF ZeichenZahl<MaxStringLaenge
                      THEN
                        INC(ZeichenZahl);
                        Puffer[ZeichenZahl]:=Zeichen
                      END;
                    ELSE
                      IF ZeichenZahl<MaxStringLaenge
                      THEN Puffer[ZeichenZahl+1]:=0C
                      END;
                      Einfuegen(Puffer,Gesamt,Woerter);
                      Zustand:=ZeichenLesen
                    END
    END; (* CASE *)
    Read(Zeichen)
  END; (* WHILE *)
  CloseInput;
  Sortiere(Woerter,1,Gesamt);
  FOR i:=1 TO Gesamt DO
    WriteCard(Woerter[i].Anzahl,5);
    WriteString(" x ");
    WriteDynString(Woerter[i].Wort);
    WriteLn
  END (* FOR *)
END WortHaeufigkeiten.
```

Probelauf: (Eingabedatei war das Definitionsmodul von DynStr ohne Kommentare)

Worthäufigkeiten in einem Text

Input from: DYNSTR.DEF
3 x ARRAY
1 x BOOLEAN
1 x CARDINAL
3 x CHAR
1 x CONST
1 x DEFINITION
2 x DynStr
8 x DynString
1 x DynStringLength
1 x DynStringLess
1 x END
1 x ForgetDynString
1 x MODULE
1 x MakeDynString
1 x MakeStatString
2 x MaxStringLaenge
3 x OF

1 x POINTER
7 x PROCEDURE
1 x ReadDynString
8 x S
4 x StatString
1 x TO
1 x TYPE
4 x VAR
1 x WriteDynString

Aufgaben:

1. Ergänzen Sie das Modul «DynStr» um eine Funktionsprozedur, die die erste Position eines statischen Teilstrings in einem dynamischen String zurückgibt. Falls der Teilstring nicht vorkommt, soll der Rückgabewert 0 sein.

   Beispiel:

```
VAR TestString : DynString;
...
MakeDynString("Modula-2",TestString);
...

DynStringPos("du",TestString) ->  3
DynStringPos("Modula",TestString) -> 1
DynStringPos("Hallo",TestString)  -> 0
```

2. Die Rückführung der Gleichheit zweier Strings auf Kleiner (DynStringLess) ist sehr rechenintensiv, da zwei komplette Vergleiche notwendig sind. Erweitern Sie das Bibliotheksmodul «DynStr» um die Funktionsprozedur «DynStringEqual», die die Gleichheit in einem Durchgang bestimmt.
3. Ergänzen Sie das Programm «WortHaeufigkeiten» um eine Ausgabeprozedur, mit der die Ausgabe auch umgeleitet werden kann.

9.3 Rekursive Datenstrukturen

Betrachten wir einmal folgende rekursive Definition:

«Eine Liste ist entweder leer, oder sie besteht aus einem Kopf, gefolgt von einer Liste.»

Eine solche Datenstruktur ist mit statischen Datentypen nicht darstellbar. Die Konstruktion

```
TYPE Liste = RECORD
               CASE leer : BOOLEAN OF
                 TRUE : (* nichts *)
                |FALSE: Kopf : IrgendEinTyp;
                        Rest : Liste
               END (* CASE *)
             END; (* RECORD *)
```

wird vom Compiler mit einer Fehlermeldung wie «unbekannter Typ» bei «Rest : Liste» zurückgewiesen. Im Gegensatz zu Prozeduren, die nach dem Prozedurkopf bereits bekannt sind (also auch im eigenen Anweisungsteil), muß eine Typdefinition erst vollständig abgeschlossen sein, ehe auf den neudefinierten Typbezeichner zugegriffen werden kann.

Da eine Variable vom Typ «Liste» beliebig viele Elemente enthalten kann, kann der benötigte Speicherbereich während des Übersetzens nicht bestimmt werden. Aus diesem Grund entzieht sich die Struktur «Liste» einer gleichen Behandlung wie statische Datentypen.

Dennoch ist der Datentyp «Liste» in Modula-2 realisierbar. Rekursive Datenstrukturen können mit Zeigertypen zusammengebaut werden:

```
TYPE Liste = POINTER TO Kopf
     Kopf= RECORD
             Inhalt : IrgendEinTyp
             Rest : Liste
           END;
```

Interessanterweise wird diese Definition vom Compiler akzeptiert, obwohl in der Definition von «Liste» auf einen Typ namens «Kopf» Bezug genommen wird und dieser Typ an dieser Stelle noch nicht bekannt ist. Wir haben es hier mit der einzigen Ausnahme der Regel «erst deklarieren, dann verwenden» zu tun:

Bei der Definition eines Pointer-Typs darf auf Typen zugegriffen werden, die erst später definiert werden.

9.3.1 Listen

Eine Liste ist offenbar eine Speicherstruktur, die jede beliebige Größe annehmen kann. Je nachdem, wie eine Liste organisiert ist, d. h. in welcher Reihenfolge Listenelemente in eine bestehende Liste eingefügt und entfernt werden, unterscheidet man zwischen

- Stapel-Strukturen, bei denen das zuletzt eingefügte Element als erstes wieder entfernt wird, und
- Puffer-Strukturen, bei denen die Reihenfolge von Einfügen und Entfernen übereinstimmt.

Wir wollen nun eine einfache Liste aufbauen. Die erste Überlegung ist, wie wir eine leere Liste darstellen können. Hierfür gibt es einen ausgezeichneten Zeiger, eine Zeigerkonstante sozusagen, mit dem Namen «NIL», der von dem lateinischen Wort «nihil» (nichts) abgeleitet wurde. Diese Konstante kann jeder Zeigervariablen zugewiesen werden. Damit später untersucht werden kann, ob ein Zeiger den Wert «NIL» (oder den eines anderen Zeigers) hat, müssen auch Vergleiche von Zeigern möglich sein. In der Tat, Zeigervariable können mit den Operatoren = und <> (bzw. #) verglichen werden.

Beispiel:

```
VAR p,q : POINTER TO CARDINAL;
...
IF p=q THEN ...
```

Sind zwei Zeigervariable gleich, so bedeutet das, daß sie auf dieselbe dynamische Variable zeigen. Deren Inhalte (p^ und q^) sind dann nicht nur in dem Sinn gleich, daß sie gleiche Werte enthalten, sie sind sogar in dem Sinn identisch, daß es sich um dieselben Speicherzellen handelt! Es ist also folgendes zu beachten:

Aus 'p=q' folgt immer 'p^ = q^' oder 'p=NIL' und 'q=NIL'.
Dieser Schluß ist jedoch nicht umkehrbar!

Beispiel:

```
VAR p,q : POINTER TO CARDINAL;
 ...
```

```
ALLOCATE(p,TSIZE(CARDINAL));
p^:=12345;
q:=p;  (* Jetzt zeigen p und q auf dieselbe Variable, es gilt
          p^ = q^ = 12345 *)
ALLOCATE(q,TSIZE(CARDINAL));
       (* q erhält dadurch einen neuen Wert *)
q^:=12345;  (* Jetzt gilt zwar wieder p^ = q^ = 12345. Die Relation
               p = q ist aber nicht mehr erfüllt *)
```

Zurück zu unserer Liste. Sie wird laut obiger Definition durch einen Zeiger auf den Listenkopf dargestellt. Der Listenkopf selbst enthält einen Zeiger auf den Rest der Liste. Ein Algorithmus zum Einfügen eines neuen Elementes am Anfang der Liste kann somit so formuliert werden:

Erzeuge einen neuen Kopf und hänge an diesen die bisherige Liste an.

```
TYPE  Liste = POINTER TO Kopf;
      Kopf  = RECORD
                Inhalt : CARDINAL;  (* oder irgendein anderer Typ *)
                Rest   : Liste
              END;

VAR  ZahlenListe, BisherigeListe : Liste;
     NeuesElement : CARDINAL;
   ...

   (* Einfügen von NeuesElement in die ZahlenListe *)
   BisherigeListe:=ZahlenListe;          (* Zwischenspeichern *)
   ALLOCATE(ZahlenListe,TSIZE(Kopf));    (* Neuen Kopf erzeugen *)
   ZahlenListe^.Inhalt:=NeuesElement;    (* Inhalt zuweisen *)
   ZahlenListe^.Rest:=BisherigeListe;    (* BisherigeListe anhängen *)
```

Das Entfernen des ersten Listenelementes (Kopf) geschieht in analoger Weise:

```
VAR ZahlenListe, ListenKopf : Liste;
    ErstesElement : CARDINAL;
  ...

  ListenKopf:=ZahlenListe;             (* Kopf abspalten *)
  ErstesElement:=ListenKopf^.Inhalt;   (* ErstesElement zuweisen *)
  ZahlenListe:=ListenKopf^.Rest;       (* Der neue Kopf ist der Rest *)
  DEALLOCATE(ListenKopf,TSIZE(Kopf)); (* Alten Kopf löschen *)
```

9.3.1.1 Stapel

Wird eine Liste auf diese Weise organisiert, daß also sowohl Einfügen als auch Entfernen nur am Listenkopf stattfinden, so spricht man von Stapel- (engl. Stack) oder LIFO-(Last In, First Out = zuletzt rein, zuerst raus)Struktur. Wir werden die Arbeit mit einem Stapel an einem kleinen Beispielprogramm demonstrieren. Dazu formulieren wir jeweils eine Einfüge- und eine Entfernprozedur sowie eine Funktion, die anzeigt, ob der Stapel leer ist oder nicht.

```
MODULE StapelDemo;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM  IMPORT TSIZE;
FROM InOut IMPORT WriteString, WriteLn, WriteCard, ReadCard;

TYPE  Liste = POINTER TO Kopf;
      Kopf  = RECORD
                Inhalt : CARDINAL;  (* oder irgendein anderer Typ *)
                Rest   : Liste
              END;

VAR  Stapel : Liste;
     Zahl : CARDINAL;

PROCEDURE Einfuegen(VAR S : Liste; NeuesElement : CARDINAL);
  VAR BisherigeListe : Liste;
  BEGIN
    BisherigeListe:=S;
    ALLOCATE(S,TSIZE(Kopf));
    S^.Inhalt:=NeuesElement;
    S^.Rest:=BisherigeListe
  END Einfuegen;

PROCEDURE Entfernen(VAR S : Liste; VAR ErstesElement : CARDINAL);
  VAR ListenKopf : Liste;
  BEGIN
    ListenKopf:=S;
    ErstesElement:=ListenKopf^.Inhalt;
    S:=ListenKopf^.Rest;
    DEALLOCATE(ListenKopf,TSIZE(Kopf))
  END Entfernen;

PROCEDURE ListeLeer(S : Liste) : BOOLEAN;
  BEGIN
    RETURN S=NIL
  END ListeLeer;

BEGIN
  WriteString("Demonstration eines Stapels"); WriteLn;
  WriteLn;
  WriteString("Bitte geben Sie eine Folge von Zahlen ein (0=Ende):");
  WriteLn;
  Stapel:=NIL;  (* Wichtig! Anfangs ist der Stapel leer *)
  REPEAT
    ReadCard(Zahl); WriteLn;
    IF Zahl>0 THEN Einfuegen(Stapel,Zahl) END
```

```
  UNTIL Zahl=0;
  WriteString("Jetzt wird der Stapel abgebaut:"); WriteLn;
  WHILE NOT ListeLeer(Stapel) DO
    Entfernen(Stapel,Zahl);
    WriteCard(Zahl,10); WriteLn
  END (* WHILE *)
END StapelDemo.
```

Testlauf:

Demonstration eines Stapels

Bitte geben Sie eine Folge von Zahlen ein (0=Ende):
12
18
22
11
123
144
16
17
0

Jetzt wird der Stapel abgebaut:
17
16
144
123
11
22
18
12

Aufgaben:

1. Was würde passieren, wenn die Anweisung «Stapel:=NIL» fehlen würde?
2. Stellen Sie das Programm so um, daß Zeichenketten eingelesen und ausgegeben werden.

9.3.1.2 Puffer

Die zweite wichtige Listenorganisation funktioniert nach dem FIFO-Prinzip, wobei das zuerst eingefügte Element (First In) auch als erstes wieder entfernt (First Out) wird. Eine solche Struktur wird Puffer (oder Queue) genannt. Der Grundalgorithmus zum Einfügen:

Erzeuge einen neuen Kopf und hänge ihn ans Ende der Liste an.

Entsprechend der rekursiven Struktur der Liste kann diese Vorschrift in einen rekursiven Algorithmus übertragen werden:

Falls die Liste leer ist,
dann mach' ihr einen neuen Kopf mit leerem Rest,
ansonsten wende dieses Verfahren auf den Rest an.

```
PROCEDURE Einfuegen(VAR P:Liste; NeuesElement:CARDINAL);
  BEGIN
    IF P=NIL
    THEN
      ALLOCATE(P,TSIZE(Kopf));
      P^.Inhalt:=NeuesElement;
      P^.Rest:=NIL
    ELSE Einfuegen(P^.Rest,NeuesElement)
    END
  END Einfuegen;
```

Allerdings ist dieses Verfahren nicht sehr effizient, die Rekursionstiefe steigt mit der Anzahl der Listenelemente. In diesem Fall ist es wesentlich günstiger, die Liste über zwei Zeiger – «Anfang» und «Ende» – zu verwalten. Am Ende wird eingefügt, am Anfang entfernt. Als einziger Sonderfall ist der leere Puffer zu beachten. Hier müssen sowohl Anfang als auch Ende initialisiert werden.

```
TYPE PufferTyp = RECORD
                   Anfang, Ende : Liste
                 END;
```

Die einzelnen Prozeduren können jetzt leicht implementiert werden:

```
MODULE PufferDemo;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM  IMPORT TSIZE;
FROM InOut IMPORT WriteString, WriteLn, WriteCard, ReadCard;
```

```
TYPE  Liste = POINTER TO Kopf;
      Kopf  = RECORD
                Inhalt : CARDINAL;  (* oder irgendein anderer Typ *)
                Rest   : Liste
              END;

     PufferTyp = RECORD
                   Anfang, Ende : Liste
                 END;

VAR  Puffer : PufferTyp;
     Zahl : CARDINAL;

PROCEDURE Einfuegen(VAR P : PufferTyp; NeuesElement : CARDINAL);
  VAR NeuesEnde : Liste;
  BEGIN
    ALLOCATE(NeuesEnde,TSIZE(Kopf));
    NeuesEnde^.Inhalt:=NeuesElement;
    NeuesEnde^.Rest:=NIL; (* Sonst wäre es kein Ende! *)
    IF P.Anfang=NIL (* Puffer leer? *)
    THEN P.Anfang:=NeuesEnde; P.Ende:=NeuesEnde
    ELSE
      P.Ende^.Rest:=NeuesEnde; (* An die Liste anhängen *)
      P.Ende:=NeuesEnde        (* und Ende entsprechend versetzen *)
    END (* IF *)
  END Einfuegen;

PROCEDURE Entfernen(VAR P : PufferTyp; VAR ErstesElement : CARDINAL);
  VAR ListenKopf : Liste;
  BEGIN
    ListenKopf:=P.Anfang;
    ErstesElement:=ListenKopf^.Inhalt;
    P.Anfang:=ListenKopf^.Rest;
    DEALLOCATE(ListenKopf,TSIZE(Kopf))
  END Entfernen;

PROCEDURE ListeLeer(P : PufferTyp) : BOOLEAN;
  BEGIN
    RETURN P.Anfang=NIL
  END ListeLeer;

BEGIN
  WriteString("Demonstration eines Puffers"); WriteLn;
  WriteLn;
  WriteString("Bitte geben Sie eine Folge von Zahlen ein (0=Ende):");
  WriteLn;
  Puffer.Anfang:=NIL;  (* Wichtig! Anfangs ist der Puffer leer *)
  REPEAT
    ReadCard(Zahl); WriteLn;
    IF Zahl>0 THEN Einfuegen(Puffer,Zahl) END
  UNTIL Zahl=0;
  WriteString("Jetzt wird der Puffer ausgelesen:"); WriteLn;
  WHILE NOT ListeLeer(Puffer) DO
    Entfernen(Puffer,Zahl);
    WriteCard(Zahl,10); WriteLn
  END (* WHILE *)
END PufferDemo.
```

Testlauf:

```
Demonstration eines Puffers

Bitte geben Sie eine Folge von Zahlen ein (0=Ende):
23
43
776
324
123
978
323
0

Jetzt wird der Puffer ausgelesen:
23
43
776
324
123
978
323
```

Aufgabe:

Es soll ein Programm geschrieben werden, das eine Meßwertverarbeitung simuliert. Dabei gehen die Meßwerte (REAL) in unregelmäßigen Abständen ein. Die Funktion «MesswertDa():BOOLEAN» gibt an, daß ein gültiger Meßwert vorliegt, der mit der Prozedur «LiesMesswert(VAR W : REAL)» eingelesen werden muß. In den Pausen, wenn keine Meßwerte vorliegen, sollen die Werte in der Reihenfolge, wie sie einkamen, bearbeitet werden. Man spricht in diesem Fall von einer gepufferten Bearbeitung. Viele Texteditoren arbeiten beispielsweise nach diesem Verfahren (die eingehenden Daten sind hier Tastendrücke).

Auf folgende Prozeduren können Sie bei der Lösung zurückgreifen:

```
FROM RANDOM IMPORT RandomCard, RandomReal;
FROM InOut IMPORT WriteString, WriteLn, WriteReal;

  PROCEDURE MesswertDa() : BOOLEAN;
    BEGIN
      RETURN RandomCard(2)=0
    END MesswertDa;
```

```
PROCEDURE LiesMesswert(VAR Wert : REAL);
  BEGIN
    WriteString("Einlesen: ");
    Wert:=100.0*RandomReal();
    WriteReal(Wert,20); WriteLn
  END LiesMesswert;

PROCEDURE BearbeiteMesswert(Wert : REAL);
  BEGIN
    WriteString("        Bearbeiten: ");
    WriteReal(Wert,20); WriteLn
  END BearbeiteMesswert;
```

Und so sollte der Ablauf des Programms in etwa aussehen:

```
RANDOM-Startwert: 0
Simulation einer Messwert-Verarbeitung
---------------------------------------

Einlesen:  4.912579957356077E1
        Bearbeiten:  4.912579957356077E1
Einlesen:  7.151385927505330E1
        Bearbeiten:  7.151385927505330E1
Einlesen:  9.304904051172707E1
        Bearbeiten:  9.304904051172707E1
Einlesen:  7.279317697228144E1
        Bearbeiten:  7.279317697228144E1
Einlesen:  3.953091684434967E1
Einlesen:  3.739872068230277E1
        Bearbeiten:  3.953091684434967E1
        Bearbeiten:  3.739872068230277E1
Einlesen:  5.147121535181236E1
        Bearbeiten:  5.147121535181236E1
Einlesen:  8.771855010660981E1
Einlesen:  4.208955223880597E1
Einlesen:  1.364605543710021E0
Einlesen:  8.324093816631130E1
Einlesen:  3.014925373134328E1
        Bearbeiten:  8.771855010660981E1
Einlesen:  2.247334754797441E1
Einlesen:  1.095948827292110E1
        Bearbeiten:  4.208955223880597E1
Einlesen:  2.375266524520255E1
 ...
```

9.3.2 Untypisierte Listen

Wir haben gesehen, daß bei den Strukturen «Stapel» und «Puffer» der Listeninhalt nur eine sekundäre Rolle spielt. In den Prozeduren tritt er nur bei den Zuweisungen auf. Es stellt sich nun die Frage, ob es möglich ist, diese Speicherstrukturen so universell zur Verfügung zu stellen, daß

- keine speziellen Typangaben benötigt werden und
- die Verwaltung der Zeigervariablen vollkommen ausgelagert wird.

Es ist ein besonderer Vorzug von Modula-2, daß diese Möglichkeit besteht, ohne auf irgendwelche Programmiertricks zurückgreifen zu müssen. Das notwendige Werkzeug finden wir in dem Modul «SYSTEM», das u. a. den Datentyp «WORD» liefert.

«WORD» ist eine Speichereinheit, deren exakte Größe (die von System zu System variiert) uns nicht weiter interessiert. Wichtig in diesem Zusammenhang ist allein die Tatsache, daß über den offenen Feldparameter «ARRAY OF WORD» Variable beliebigen Datentyps übergeben werden können! Innerhalb der Prozedur stellt sich eine übergebene Variable dann schlicht als Speicherbereich dar, über dessen Struktur und Typ nichts bekannt ist.

Beispiel:

```
PROCEDURE WortZahl(Variable : ARRAY OF WORD) : CARDINAL;
  BEGIN
    RETURN HIGH(Variable)+1
  END WortZahl;

VAR Z : CARDINAL;
    R : REAL;
    S : ARRAY[0..9] OF CHAR;

...

WriteCard(WortZahl(Z),10)   (* Ergibt normalerweise 1 *)
WriteCard(WortZahl(R),10)   (* Häufig 4 oder 6        *)
WriteCard(WortZahl(S),10)   (* Wahrscheinlich 10      *)

...
```

Damit ist es beispielsweise möglich, Prozeduren mit untypisierten Parametern zu erstellen. Untypisiert heißt in diesem Zusammenhang, daß beim Aufruf Variable jedes beliebigen Typs eingesetzt werden können.

Beispiele:

```
PROCEDURE FillChar(VAR Variable : ARRAY OF WORD;
                   Anzahl, Konstante : CARDINAL);
  (* Füllt bei einer Variablen Anzahl Worte mit Konstante *)
  VAR i : CARDINAL;
  BEGIN
    FOR i:=0 TO Anzahl-1 DO Variable[i]:=Konstante END
  END FillChar;
```

```
PROCEDURE Move(Quelle : ARRAY OF WORD; VAR Ziel : ARRAY OF WORD;
               Anzahl : CARDINAL);
  (* Kopiert Anzahl Worte von Quelle nach Ziel *)
  VAR i : CARDINAL;
  BEGIN
    FOR i:=0 TO Anzahl-1 DO Ziel[i]:=Quelle[i] END
  END Move;
```

Die Prozedur "Move" kann als universelle Zuweisung verwendet werden:

```
VAR Zahl1, Zahl2 : REAL;
...
Move(Zahl1,Zahl2,TSIZE(REAL)) hat dieselbe Funktion wie
Zahl2:=Zahl1
```

Mit dem Typ «WORD» ist es also ohne weiteres möglich, einen untypisierten Listentyp zu schaffen. Ähnlich wie in dem Modul «DynStr» wird nur der jeweils benötigte Speicherplatz als «ARRAY[0..x] OF WORD» reserviert. Da die aktuelle Größe nicht durch eine spezielle Marke bestimmt werden kann, legen wir sie in «AnzahlWords» ebenfalls mit ab. Unsere universelle Liste hat nun folgende Struktur:

```
CONST MaxWords = 65535; (* =MAX(CARDINAL) *)
TYPE  MemArray = ARRAY[0..MaxWords] OF WORD;
      Liste = POINTER TO Kopf;
      Kopf =  RECORD
                AnzahlWords : CARDINAL;
                Inhalt : POINTER TO MemArray;
                Rest : Liste
              END;
```

Um Anwenderprogramme von dieser komplizierten Struktur freizuhalten, bedienen wir uns des opaken (unsichtbaren) Exports. Damit können Zeigertypen ohne Typdefinition im Definitionsteil eines Moduls aufgeführt sein, die Konkretisierung erfolgt erst im Implementationsteil. Im Definitionsteil steht dann nur eine Deklaration wie folgt:

```
TYPE FIFO;
...
```

Ein solcher Name wird nur als Ganzes exportiert, die Details seiner Implementierung bleiben dem importierenden Programm verborgen. Dadurch wird jede Möglichkeit illegaler Zugriffe wirksam unterbunden. Hinweis: Der opake Export ist meist auf Zeigertypen beschränkt!

```
DEFINITION MODULE LIFOLib;
  (* Bibliotheks-Modul für eine universelle Stapel-Verwaltung *)

  FROM SYSTEM IMPORT WORD;

  TYPE LIFO; (* Opaker Export eines Zeigertyps. Die Details werden im
                Implementations-Modul versteckt *)

  PROCEDURE InitLIFO(VAR L : LIFO);
  (* Initialisiert eine LIFO-Struktur *)

  PROCEDURE EmptyLIFO(L : LIFO):BOOLEAN;
  (* Gibt an, ob die LIFO-Struktur leer ist *)

  PROCEDURE PopFromLIFO(VAR L : LIFO; VAR Inh : ARRAY OF WORD);
  (* Holt das oberste Element von dem LIFO-Speicher. Falls Speicher- und
     Variablengröße nicht übereinstimmen, erfolgt eine Fehlermeldung *)

  PROCEDURE PushToLIFO(VAR L : LIFO; Inh : ARRAY OF WORD);
  (* Legt die Variable Inh auf dem LIFO-Speicher ab *)

END LIFOLib.
```

Der Implementationsteil unterscheidet sich nur sehr wenig von den bisher bekannten Prozeduren für Stapelspeicher. Anstelle der Inhaltszuweisung tritt der Prozeduraufruf von «Move».

```
IMPLEMENTATION MODULE LIFOLib;

  FROM Terminal IMPORT WriteString, WriteLn;
  FROM Storage IMPORT ALLOCATE, DEALLOCATE;
  FROM SYSTEM IMPORT TSIZE;

  CONST MaxWords = 65535;
  TYPE MemArray = ARRAY[0..MaxWords] OF WORD;
       Liste = POINTER TO Kopf;
       Kopf = RECORD
                Anzahl : [0..MaxWords];
                Inhalt : POINTER TO MemArray;
                Rest : Liste
              END;
       LIFO = Liste;

  PROCEDURE Move(Quelle : ARRAY OF WORD; VAR Ziel : ARRAY OF WORD;
                 Anzahl : CARDINAL);
    (* Kopiert Anzahl Worte von Quelle nach Ziel *)
    VAR i : CARDINAL;
    BEGIN
      FOR i:=0 TO Anzahl-1 DO Ziel[i]:=Quelle[i] END
    END Move;

  PROCEDURE InitLIFO(VAR L : LIFO);
    BEGIN
      L.Anfang:=NIL;
    END InitLIFO;
```

```
PROCEDURE EmptyLIFO(L : LIFO):BOOLEAN;
  BEGIN
    RETURN L.Anfang=NIL
  END EmptyLIFO;

PROCEDURE PopFromLIFO(VAR L : LIFO; VAR Inh : ARRAY OF WORD);
  VAR p : Liste;
  BEGIN
    IF NOT EmptyLIFO(L)
    THEN
      p:=L; L=L^.Naechster;
      WITH p^ DO
        IF Anzahl<>HIGH(Inh)
        THEN WriteLn; WriteString("Unpassende Typen!"); WriteLn; HALT
        END; (* IF *)
        Move(Inhalt^,Inh,Anzahl+1);
        DEALLOCATE(Inhalt,Anzahl+1)
      END; (* WITH *)
      DEALLOCATE(p,TSIZE(Kopf))
    END (* IF *)
  END PopFromLIFO;

PROCEDURE PushToLIFO(VAR L : LIFO; Inh : ARRAY OF WORD);
  VAR p : Liste;
  BEGIN
    ALLOCATE(p,TSIZE(Kopf));
    WITH p^ DO
      Anzahl:=HIGH(Inh);
      ALLOCATE(Inhalt,Anzahl+1);
      Move(Inh,Inhalt^,Anzahl+1);
      Naechster:=L
    END; (* WITH *)
    L:=p
    END (* IF *)
  END PushToLIFO;

END LIFOLib.
```

In analoger Weise kann ein Modul «FIFOLib» erstellt werden, das eine universelle Pufferstruktur einschließlich der benötigten Prozeduren zum Ablegen (PutToFIFO) und Entfernen (GetFromFIFO) bereitstellt.

```
DEFINITION MODULE FIFOLib;
  (* Bibliotheks-Modul für eine universelle Puffer-Verwaltung *)

  FROM SYSTEM IMPORT WORD;

  TYPE FIFO; (* Opaker Export *)

  PROCEDURE InitFIFO(VAR F : FIFO);
  (* Initialisiert den Puffer F *)

  PROCEDURE ExitFIFO(VAR F : FIFO);
  (* Deinitialisiert den Puffer F *)

  PROCEDURE EmptyFIFO(F : FIFO):BOOLEAN;
  (* Gibt an, ob der Puffer F leer ist oder nicht *)
```

```
  PROCEDURE GetFromFIFO(VAR F : FIFO; VAR Inh : ARRAY OF WORD);
  (* Holt das erste Element des Puffers in die Variable Inh.
     Falls die Größen nicht übereinstimmen, wird das Programm mit einer
     Fehlermeldung abgebrochen *)

  PROCEDURE PutToFIFO(VAR F : FIFO; Inh : ARRAY OF WORD);
  (* Legt die Variable Inh im Puffer ab *)

END FIFOLib.
```

Die im Implementations-Modul versteckte Struktur von «FIFO» ist etwas komplizierter als erwartet. Der Grund liegt darin, daß der opake Export ja nur mit Zeigertypen möglich ist. Aus diesem Grund ist ein FIFO nicht einfach ein RECORD mit den Komponenten «Anfang» und «Ende», sondern ein Zeiger auf eine solche Struktur. Deshalb gibt es neben «InitFIFO» auch noch die Prozedur «ExitFIFO», die den belegten Speicherplatz wieder freigibt.

```
IMPLEMENTATION MODULE FIFOLib;

  FROM Terminal IMPORT WriteString, WriteLn;
  FROM STORAGE IMPORT ALLOCATE, DEALLOCATE;
  FROM SYSTEM IMPORT TSIZE;

  CONST MaxWords = 65535;
  TYPE MemArray = ARRAY[0..MaxWords] OF WORD;
       Liste = POINTER TO Kopf;
       Kopf = RECORD
                Anzahl : [0..MaxWords];
                Inhalt : POINTER TO MemArray;
                Rest : Liste
              END;
       FIFORec = RECORD
                   Anfang, Ende : Liste
                 END;
       FIFO = POINTER TO FIFORec;

  PROCEDURE Move(Quelle : ARRAY OF WORD; VAR Ziel : ARRAY OF WORD;
                 Anzahl : CARDINAL);
    (* Kopiert Anzahl Worte von Quelle nach Ziel *)
    VAR i : CARDINAL;
    BEGIN
      FOR i:=0 TO Anzahl-1 DO Ziel[i]:=Quelle[i] END
    END Move;

  PROCEDURE InitFIFO(VAR F : FIFO);
    BEGIN
      ALLOCATE(F,TSIZE(FIFORec));
      F^.Anfang:=NIL;
    END InitFIFO;

  PROCEDURE ExitFIFO(VAR F : FIFO);
    BEGIN
      DEALLOCATE(F,TSIZE(FIFORec));
      F:=NIL
    END ExitFIFO;
```

```
PROCEDURE EmptyFIFO(F : FIFO):BOOLEAN;
  BEGIN
    RETURN F^.Anfang=NIL
  END EmptyFIFO;

PROCEDURE GetFromFIFO(VAR F : FIFO; VAR Inh : ARRAY OF WORD);
  VAR p : Liste;
      i : [0..MaxWords];
  BEGIN
    IF NOT EmptyFIFO(F)
    THEN
      p:=F^.Anfang; F^.Anfang:=F^.Anfang^.Naechster;
      WITH p^ DO
        IF Anzahl<>HIGH(Inh)
        THEN WriteLn; WriteString("Unpassende Typen!"); WriteLn; HALT
        END; (* IF *)
        Move(Inhalt^,Inh,Anzahl+1);
        DEALLOCATE(Inhalt,Anzahl+1)
      END; (* WITH *)
      DEALLOCATE(p,TSIZE(Kopf))
    END (* IF *)
  END GetFromFIFO;

PROCEDURE PutToFIFO(VAR F : FIFO; Inh : ARRAY OF WORD);
  VAR p : Liste;
      i : [0..MaxWords];
  BEGIN
    ALLOCATE(p,TSIZE(Kopf));
    WITH p^ DO
      Anzahl:=HIGH(Inh);
      ALLOCATE(Inhalt,Anzahl+1);
      Move(Inh,Inhalt^,Anzahl+1);
      Naechster:=NIL
    END; (* WITH *)
    IF EmptyFIFO(F)
    THEN F^.Anfang:=p; F^.Ende:=p
    ELSE F^.Ende^.Naechster:=p; F^.Ende:=p
    END (* IF *)
  END PutToFIFO;

END FIFOLib.
```

Aufgabe:

Die Meßwert-Verarbeitung (Aufgabe 7.3.1.2) soll so abgeändert werden, daß die aktuellsten Meßwerte zuerst bearbeitet werden. Verwenden Sie zur Lösung das Bibliotheksmodul «LIFOLib».

9.3.3 Bäume

Prinzipiell kann eine Liste auch so organisiert werden, daß ihre Elemente in geordneter Reihenfolge vorliegen. Man spricht dann von einer sortierten Liste. Da allerdings im Mittel stets die halbe Liste durchsucht werden muß, ehe die passende Stelle zum Einfü-

gen gefunden wird, ist eine derartige Struktur nicht besonders effizient. Der große Vorteil eines geordneten Feldes in Verbindung mit der binären Suche ist bei einer sortierten Liste nicht gegeben.

Dennoch kann eine dynamische Struktur gebildet werden, die sich in bezug auf das Suchen ähnlich gut verhält wie ein geordnetes Feld, es beim Einfügen neuer Elemente sogar bei weitem Übertrifft. Es handelt sich dabei um sogenannte Bäume, die auch als Verallgemeinerung der Listen aufgefaßt werden können. Dabei ist ein Baum wie folgt definiert:

Ein Baum ist entweder leer, oder er besteht aus einem Knoten mit einem oder mehreren (Teil-)Bäumen.

Enthält jeder Knoten nur einen Teilbaum, so haben wir es wiederum mit einer Liste zu tun. Eine Liste präsentiert sich somit als spezielle (degenerierte) Form eines Baumes.

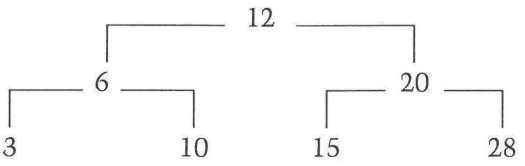
Von besonderem Interesse sind die Bäume, deren Knoten genau zwei Teilbäume enthalten. Sie werden als «Zweiwegbäume» oder «binäre Bäume» bezeichnet. Die beiden Teilbäume erhalten meist die Namen «linker» und «rechter» Teilbaum. Wir werden uns in der Folge ausschließlich mit binären Bäumen beschäftigen. Die Grundstruktur eines binären Baumes:

```
TYPE  Baum = POINTER TO Knoten;
      Knoten = RECORD
                  Inhalt : IrgendEinTyp;
                  links, rechts : Baum
               END;
```

Wenn man nun noch axiomatisch fordert, daß alle Elemente des linken Teilbaums (eines Knotens) bezüglich einer Ordnung vor, die des rechten Teilbaums nach dem Knoteninhalt kommen, so spricht man von einem geordneten binären Baum bzw. von einem binären Suchbaum.

9.3.3.1 Binäre Suchbäume

Betrachten wir beispielsweise einen Baum, dessen Inhalte CARDINAL-Zahlen sind. Der erste Knoten (bei Listen war das der Kopf) wird normalerweise mit «Wurzel» bezeichnet. Man kann sich einen binären Suchbaum am einfachsten so vorstellen, daß die Wurzel oben ist und die Zweige nach unten zeigen.



Die Suche nach einem Element ist recht einfach. Man beginnt bei der Wurzel. Ist das gesuchte Element kleiner als deren Inhalt, so ist die Suche im linken Teilbaum fortzusetzen. Ist es größer, so muß es sich im rechten Teilbaum befinden. Andernfalls ist des gesuchte Element gefunden. Wird bei der Suche das Ende eines Astes (durch NIL gekennzeichnet) erreicht, so ist das Element nicht vorhanden.

9.3.3.2 Einfügen, Suchen und Löschen im binären Suchbaum

Das Einfügen eines neuen Elementes kann unmittelbar aus der Suche gewonnen werden. Wenn es nicht schon vorhanden ist, ist sein Platz das Ende des Astes, das bei der Suche erreicht wird.

Bei der Formulierung der benötigten Prozeduren zum Einfügen und Suchen gehen wir von folgender Typdefinition aus:

```
TYPE Baum = POINTER TO Knoten;
     Knoten = RECORD
                Inhalt : CARDINAL;
                links, rechts : Baum
              END;
```

Als Ordnungsrelation verwenden wir die Ordnung der natürlichen Zahlen.

```
PROCEDURE Vorhanden(B : Baum; Zahl : CARDINAL) : BOOLEAN;
  (* Gibt an, ob die Zahl im Baum B vorhanden ist *)
  VAR p : Baum;  (* Ein Hilfszeiger zum Durchsuchen des Baumes *)
  BEGIN
  p:=B;
  WHILE p<>NIL DO
    IF Zahl<p^.Inhalt    (* ist die gesuchte Zahl kleiner?  *)
    THEN p:=p^.links     (* im linken Teilbaum weitersuchen *)
    ELSIF Zahl>p^.Inhalt (* oder ist sie größer             *)
    THEN p:=p^.rechts    (* dann im rechten Teilbaum suchen *)
    ELSE RETURN TRUE     (* ansonsten ist sie gefunden      *)
    END
  END; (* WHILE *)
  (* Das Ende der Suche ist erfolglos erreicht *)
  RETURN FALSE
END Vorhanden;
```

Hinweis: Mit jedem Schritt verdoppelt sich die Anzahl der untersuchten Knoten (bei einem ausgeglichenen Baum). Dadurch ent-

spricht der Suchaufwand dem der binären Suche in einem geordneten Feld (mit gleicher Elementzahl).

Beim Einfügen des Baumes benötigen wir einen weiteren Hilfszeiger «q», der auf den zuletzt betrachteten Knoten zeigt, sowie einen Richtungszeiger, der festhält, ob wir von diesem Knoten nach links oder rechts weitergesucht haben.

```
PROCEDURE Einfügen(VAR B : Baum; Zahl : CARDINAL);
(* Fügt die Zahl in den Baum ein, wenn sie noch nicht vorhanden ist *)
  VAR p,q : Baum;
      Richtung : (l,r);
  BEGIN
    p:=B; q:=NIL;
    WHILE p<>NIL DO
      q:=p;                              (* q zeigt auf den letzten Knoten *)
      IF Zahl<p^.Inhalt                  (* ist die Zahl kleiner?          *)
      THEN p:=p^.links; Richtung:=l      (* dann links weitersuchen        *)
      ELSIF Zahl>p^.Inhalt               (* oder größer?                   *)
      THEN p:=p^.rechts; Richtung:=r     (* dann eben rechts               *)
      ELSE RETURN                        (* Die Zahl ist schon vorhanden!  *)
    END;
    (* p zeigt jetzt auf NIL und hat keine Funktion mehr. Deshalb kann  *)
    (* es als Hilfsvariable zur Erzeugung einer neuen dynamischen       *)
    (* Variablen verwendet werden.                                      *)
    ALLOCATE(p,TSIZE(Knoten));
    p^.Inhalt:=Zahl; p^.links:=NIL; p^.rechts:=NIL;
    IF q=NIL                             (* war der Baum leer?          *)
    THEN B:=p                            (* dann ist p die Wurzel       *)
    ELSIF Richtung=l                     (* vom letzten Knoten nach links?*)
    THEN q^.links:=p
    ELSE q^.rechts:=p                    (* oder rechts                 *)
  END Einfuegen;
```

Wie schon bei den Listen, kann auch das Einfügen eines neuen Elementes in die rekursive Struktur eines binären Suchbaumes rekursiv formuliert werden:

Ist ein Baum leer, so ist an dessen Stelle der neue Knoten zu setzen, andernfalls, wenn das neue Element kleiner als der Bauminhalt ist, ist es im linken Teilbaum einzufügen, andernfalls, wenn es größer als der Bauminhalt ist, ist es im rechten Teilbaum einzufügen, andernfalls ist es schon vorhanden.

```
PROCEDURE Einfuegen(VAR B : Baum; Zahl : CARDINAL);
  BEGIN
    IF B=NIL
    THEN
      ALLOCATE(B,TSIZE(Knoten));
      B^.Inhalt:=Zahl; B^.links:=NIL; B^.rechts:=NIL
    ELSIF Zahl<B^.Inhalt
    THEN Einfuegen(B^.links,Zahl)
    ELSIF Zahl>.B^.Inhalt
    THEN Einfuegen(B^.rechts,Zahl)
  END Einfuegen;
```

Dieser rekursive Algorithmus besticht durch seine Einfachheit. Da zudem die Rekursionstiefe die Baumhöhe niemals überschreitet, ist kein Effizienzverlust gegenüber der nicht-rekursiven Fassung zu verzeichnen.

Am kompliziertesten gestaltet sich das Löschen eines Knotens. Der Algorithmus lautet:

Wenn es sich bei dem zu löschenden Knoten um einen Endknoten (mit zwei leeren Teilbäumen) handelt, so kann er, nachdem der Verweis auf ihn gelöscht (auf NIL gesetzt) wurde, einfach entfernt werden.

Andernfalls,
wenn der linke Teilbaum des zu löschenden Knotens nicht leer ist, dann ersetze seinen Inhalt durch den des größten Knotens des linken Teilbaums und entferne diesen,
andernfalls
ersetze seinen Inhalt durch den des kleinsten Knotens des rechten Teilbaums und entferne diesen.

Bei der Umsetzung dieser Vorschrift bedienen wir uns der Hilfszeiger p, q und o.

p –> zeigt auf den zu löschenden Knoten,
q –> zeigt auf den zu entfernenden Knoten (wenn es sich bei p nicht um einen Endknoten handelt),
o –> zeigt auf den Vorgänger von q.

```
PROCEDURE Loeschen(VAR B : Baum; Zahl : CARDINAL);
  VAR p,q,o : Baum;
      Richtung : (l,r);
      gefunden : BOOLEAN;
  BEGIN
    p:=B; o:=NIL; gefunden:=FALSE;
    WHILE (p<>NIL) AND NOT gefunden DO
      IF Zahl<p^.Inhalt
      THEN o:=p; p:=p^.links; Richtung:=l;
      ELSIF Zahl>p^.Inhalt
      THEN o:=p; p:=p^.rechts; Richtung:=r
      ELSE gefunden:=TRUE
      END (* IF *)
    END; (* WHILE *)
    IF NOT gefunden THEN RETURN END;
    (* Jetzt zeigt p auf den zu löschenden Knoten *)
    IF p^.links<>NIL  (* Linker Teilbaum ist nicht leer *)
    THEN
      o:=p; q:=p^.links;
      WHILE q^.rechts<>NIL DO o:=q; q:=q^.rechts END;
      (* q zeigt auf den größten Knoten des linken Teilbaums *)
      p^.Inhalt:=q^.Inhalt;
```

```
    IF o=p THEN p^.links:=q^.links ELSE o^.rechts:=q^.links END;
    DEALLOCATE(q,TSIZE(Knoten))
  ELSIF p^.rechts<>NIL
  THEN
    o:=p; q:=p^.rechts;
    WHILE q^.links<>NIL DO o:=q; q:=q^.links END;
    p^.Inhalt:=q^.Inhalt;
    IF o=p THEN p^.rechts:=q^.rechts ELSE o^.links:=q^.rechts END;
    DEALLOCATE(q,TSIZE(Knoten))
  ELSE (* p zeigt auf einen Endknoten *)
    DEALLOCATE(p,TSIZE(Knoten));
    IF o=NIL THEN B:=NIL
    ELSIF Richtung=l THEN o^.links:=NIL
    ELSE o^.rechts:=NIL
    END
  END (* IF *)
END Loeschen;
```

9.3.3.3 Ausgabe eines Baumes

Spielerisch einfach im Gegensatz zum Löschen eines Knotens kann die sortierte Ausgabe eines Baumes programmiert werden. Wenn die einzelnen Knoten in aufsteigender Reihenfolge ausgegeben werden sollen, wird erst der linke Teilbaum, dann der Knoten selbst und schließlich der rechte Teilbaum ausgegeben. Bei absteigender Reihenfolge wird erst der rechte Teilbaum, dann der Knoten selbst und anschließend der linke Teilbaum ausgegeben. Hier gibt es keinen vernünftigen Ersatz für die rekursive Formulierung:

```
PROCEDURE Ausgabe(B : Baum);  (* Aufsteigende Ausgabe *)
  BEGIN
    IF B<>NIL
    THEN
      Ausgabe(B^.links);
      WriteCard(B^.Inhalt,10); WriteLn;
      Ausgabe(B^.rechts)
    END (* IF *)
  END Ausgabe;
```

Als Programmbeispiel wollen wir nochmals das Problem des Wörterzählens aufgreifen. Anstelle eines Feldes, in dem die Wortvorkommen gespeichert werden, tritt jetzt ein binärer Suchbaum. Damit fallen nun endgültig alle Beschränkungen weg:

- Es können beliebig viele Wörter abgelegt werden (solange der Hauptspeicher des Computers ausreicht).
- Die Frage, ob ein Wort bereits vorhanden ist, kann aufgrund der Baumstruktur sehr schnell beantwortet werden.
- Die alphabetische Ordnung ergibt sich ganz automatisch.

```
MODULE WoerterZaehlen;

  FROM DynStr IMPORT StatString, DynString, MakeDynString, MakeStatString,
                     ForgetDynString, DynStringLess, WriteDynString,
                     MaxStringLaenge;

  FROM InOut IMPORT Read, OpenInput, CloseInput, Done, WriteCard,
                    WriteString, WriteLn, OpenOutput, CloseOutput;

  FROM Storage IMPORT ALLOCATE;

  FROM SYSTEM IMPORT TSIZE;

  TYPE WortVorkommen = RECORD
                         Wort : DynString;
                         Anzahl : CARDINAL
                       END;
       Baum = POINTER TO Knoten;
       Knoten = RECORD
                  Inhalt : WortVorkommen;
                  links, rechts : Baum
                END;
       WortListe = Baum;

  VAR Woerter : WortListe;
      Puffer : StatString;
      ZeichenZahl : [0..MaxStringLaenge];
      Zeichen : CHAR;
      Zustand : (ZeichenLesen, WortLesen);

  PROCEDURE Einfuegen(S : StatString; VAR B : WortListe);
    VAR temp : DynString;
        p, q : Baum;
        Richtung : (l,r);
    BEGIN
      MakeDynString(S,temp);
      p:=B; q:=NIL;
      WHILE p<>NIL DO
        q:=p;
        IF DynStringLess(temp,p^.Inhalt.Wort)
        THEN p:=p^.links; Richtung:=l
        ELSIF DynStringLess(p^.Inhalt.Wort,temp)
        THEN p:=p^.rechts; Richtung:=r
        ELSE ForgetDynString(temp); INC(p^.Inhalt.Anzahl); RETURN
        END (* IF *)
      END; (* WHILE *)
      ALLOCATE(p,TSIZE(Knoten));
      WITH p^ DO
        links:=NIL; rechts:=NIL;
        WITH Inhalt DO Wort:=temp; Anzahl:=1 END;
      END; (* WITH p^ *)
      IF q=NIL THEN B:=p
      ELSIF Richtung=l THEN q^.links:=p
      ELSE q^.rechts:=p
      END (* IF *)
    END Einfuegen;

  PROCEDURE Ausgabe(B : WortListe);
    BEGIN
      IF B<>NIL
      THEN WITH B^ DO
```

```
            Ausgabe(links);
            WITH Inhalt DO
              WriteCard(Anzahl,5);
              WriteString(" x ");
              WriteDynString(Wort);
              WriteLn
            END; (* WITH Inhalt *)
            Ausgabe(rechts)
          END (* WITH B^ *)
      END (* IF *)
    END Ausgabe;

BEGIN (* WoerterZaehlen *)
  WriteString("Worthäufigkeiten in einem Text"); WriteLn;
  WriteString("------------------------------"); WriteLn;
  WriteLn;
  OpenInput(""); IF NOT Done THEN WriteString("Keine Datei!"); HALT END;
  Woerter:=NIL; Zustand:=ZeichenLesen;
  Read(Zeichen);
  WHILE Done DO
    CASE Zustand OF
      ZeichenLesen : IF (CAP(Zeichen)>='A') AND (CAP(Zeichen)<='Z')
                     THEN
                       ZeichenZahl:=0;
                       Puffer[ZeichenZahl]:=Zeichen;
                       Zustand:=WortLesen
                     END
    | WortLesen    : IF (CAP(Zeichen)>='A') AND (CAP(Zeichen)<='Z')
                     THEN
                       IF ZeichenZahl<MaxStringLaenge
                       THEN
                         INC(ZeichenZahl);
                         Puffer[ZeichenZahl]:=Zeichen
                       END;
                     ELSE
                       IF ZeichenZahl<MaxStringLaenge
                       THEN Puffer[ZeichenZahl+1]:=0C
                       END;
                       Einfuegen(Puffer,Woerter);
                       Zustand:=ZeichenLesen
                     END
    END; (* CASE *)
    Read(Zeichen)
  END; (* WHILE *)
  CloseInput;
  OpenOutput(""); Ausgabe(Woerter); CloseOutput
END WoerterZaehlen.
```

9.4 Baum, Liste und dynamische Strings: Die Querverweisliste

Zum Abschluß des Kurses soll noch eine Erweiterung des obigen Programms vorgestellt werden. Es handelt sich um die Erstellung einer sogenannten Querverweisliste. Auch hier werden die Wortvorkommen analysiert. Allerdings wird nicht die Anzahl der Vorkommen gespeichert, sondern die jeweiligen Zeilennummern. Diese werden in einer FIFO-Liste abgelegt. Aus diesem Grund hat die Datenstruktur «WortVorkommen» nun folgende Struktur:

```
TYPE WortVorkommen = RECORD
                       Wort : DynString;
                       Vorkommen : FIFO
                     END;
```

Die Prozedur «Einfügen» unterscheidet sich nur dadurch von der obigen, daß – wenn das Wort bereits vorhanden ist – die übergebene Zeilennummer in der Liste der Vorkommen abgelegt wird (PutTo-FIFO(p∧.Vorkommen,Nr)), ansonsten der Puffer initialisiert und die Zeilennummer abgelegt wird.

Ein wesentlich höherer Aufwand wurde bei der Ausgabe betrieben (eine halbwegs vernünftig formatierte Ausgabe von Listen, Tabellen etc. gehört zu den nerven- und zeitraubendsten Arbeiten eines Programmierers). Die drei Konstanten «Spalten», «Wort-Breite» und «SpaltenProNr» sind auf das jeweilige Ausgabemedium abzustimmen.

Spalten	–>	Anzahl der Zeichen, die in einer Zeile ausgegeben werden können, ohne daß ein automatischer Zeilenvorschub stattfindet.
WortBreite	–>	Die Größe des Feldes, innerhalb dessen die einzelnen Wörter ausgegeben werden. Benötigt ein Wort mehr Platz, so wird der Zeilenumbruch automatisch korrigiert.
SpaltenProNr	–>	Breite des Feldes für die Ziffern und das trennende Komma.

Bei der Ausgabe eines Wortvorkommen wird nun zunächst das Wort in die Zeile geschrieben (WriteDynString(Wort)). Anschließend wird der Rest des Feldes mit Punkten aufgefüllt. Dann wird die Anzahl der Zeilennummern berechnet, die noch in dieser Zeile stehen dürfen. Bei jeder ausgegebenen Nummer wird diese Zahl um

eins erniedrigt. Ist der Wert 0 erreicht, so wird eine neue Zeile begonnen und hier, anstelle des Wortes, ein entsprechender Leerstring gedruckt. Schließlich wird die Anzahl der freien Positionen erneut berechnet.

Da das Programm insbesondere zur Analyse und Dokumentation von Modula-2-Quelltexten verwendet werden soll, werden Kommentare und Zeichenketten übersprungen. Aus diesem Grund weist der eingesetzte Automat eine starke Ähnlichkeit zu dem in «Quelltextlister» verwendeten auf. Um den Zugriff auf einzelne RECORD-Komponenten und dynamische Variable unterscheiden zu können, lassen wir bei den untersuchten Wörtern folgende Syntax zu:

Wort::=Bezeichner{"^"}{"."Bezeichner{"^"}}.
Bezeichner::=Buchstabe{Buchstabe|Ziffer}.

Deshalb wird, falls im Zustand «WortLesen» das Zeichen «.» gelesen wird, noch das nächste Zeichen untersucht. Handelt es sich hierbei wieder um einen Buchstaben, so werden beide Zeichen (Punkt und Naechstes) an den Puffer angehängt. Andernfalls wird das Zeichen wieder zurückgeschrieben und der Puffer in die Wortliste eingefügt.

Das Programm «QuerverweisListe» wurde mit unserem Quelltextlister ausgedruckt und hat sich anschließend selbst bearbeitet.

```
 1:MODULE QuerverweisListe;
 2:
 3:  FROM STORAGE IMPORT ALLOCATE;
 4:
 5:  FROM SYSTEM IMPORT TSIZE;
 6:
 7:  FROM FIFO IMPORT FIFO, EmptyFIFO, InitFIFO, PutToFIFO, GetFromFIFO;
 8:
 9:  FROM InOut IMPORT Read, Write, WriteLn, WriteCard, WriteString,
10:                   OpenInput, CloseInput, OpenOutput, CloseOutput,
11:                   Done, EOL;
12:
13:  FROM DynStr IMPORT DynString, StatString, MakeDynString, WriteDynString,
14:                    ForgetDynString, DynStringLess, DynStringLength,
15:                    MaxStringLaenge;
16:
17:  CONST Spalten = 79;
18:        WortBreite = 30;
19:        SpaltenProNr = 5;
20:
21:  TYPE WortVorkommen = RECORD
22:                         Wort : DynString;
23:                         Vorkommen : FIFO
24:                       END;
```

```
25:       Baum = POINTER TO Knoten;
26:       Knoten = RECORD
27:                  Inhalt : WortVorkommen;
28:                  links, rechts : Baum
29:                END;
30:       WortListe = Baum;
31:
32:  VAR  Woerter : WortListe;
33:
34:  PROCEDURE Eingabe(VAR B : WortListe);
35:    VAR Puffer : StatString;
36:        Zeichen, Naechstes : CHAR;
37:        Zustand : (ZeichenLesen, WortLesen,
38:                   String1, String2, Kommentar);
39:        ZeilenNr, ZeichenZahl, KommentarTiefe : CARDINAL;
40:
41:    PROCEDURE Einfuegen(S : StatString; VAR B : WortListe;
42:                        Nr : CARDINAL);
43:      VAR temp : DynString;
44:          p,q : WortListe;
45:          Richtung : (l,r);
46:      BEGIN
47:        MakeDynString(S,temp);
48:        p:=B; q:=NIL;
49:        WHILE p<>NIL DO
50:          q:=p;
51:          IF DynStringLess(temp,p^.Inhalt.Wort)
52:          THEN p:=p^.links; Richtung:=l
53:          ELSIF DynStringLess(p^.Inhalt.Wort,temp)
54:          THEN p:=p^.rechts; Richtung:=r
55:          ELSE
56:            ForgetDynString(temp);
57:            PutToFIFO(p^.Inhalt.Vorkommen,Nr);
58:            RETURN
59:          END (* IF *)
60:        END; (* WHILE *)
61:        ALLOCATE(p,TSIZE(Knoten));
62:        WITH p^ DO
63:          links:=NIL; rechts:=NIL;
64:          WITH Inhalt DO
65:            Wort:=temp; InitFIFO(Vorkommen); PutToFIFO(Vorkommen,Nr)
66:          END (* WITH Inhalt *)
67:        END; (* WITH p^ *)
68:        IF q=NIL THEN B:=p
69:        ELSIF Richtung=l THEN q^.links:=p
70:        ELSE q^.rechts:=p
71:        END (* IF *)
72:      END Einfuegen;
73:
74:    (*****************************************************************)
75:    (*            Lokales Modul zum gepufferten Lesen                *)
76:    (*****************************************************************)
77:
78:    MODULE GepuffertesLesen;
79:
80:      IMPORT Read;
81:      EXPORT ReadChar, PushBack;
82:
83:      VAR ZeichenPuffer : CHAR;
84:
```

```
85:       PROCEDURE ReadChar(VAR Zeichen : CHAR);
86:         BEGIN
87:           IF ZeichenPuffer=0C
88:           THEN Read(Zeichen)
89:           ELSE Zeichen:=ZeichenPuffer; ZeichenPuffer:=0C
90:           END (* IF *)
91:         END ReadChar;
92:
93:       PROCEDURE PushBack(Zeichen : CHAR);
94:         BEGIN
95:           ZeichenPuffer:=Zeichen
96:         END PushBack;
97:
98:       BEGIN
99:         ZeichenPuffer:=0C
100:      END GepuffertesLesen;
101:
102:    (*****************************************************************)
103:    (*                    Ende des lokalen Moduls                    *)
104:    (*****************************************************************)
105:
106:    PROCEDURE Anhaengen(Zeichen : CHAR);
107:      BEGIN
108:        IF ZeichenZahl<MaxStringLaenge
109:        THEN INC(ZeichenZahl); Puffer[ZeichenZahl]:=Zeichen
110:        END (* IF *)
111:      END Anhaengen;
112:
113:    PROCEDURE PufferEinfuegen;
114:      BEGIN
115:        IF ZeichenZahl<MaxStringLaenge
116:        THEN Puffer[ZeichenZahl+1]:=0C
117:        END; (* IF *)
118:        Einfuegen(Puffer,Woerter,ZeilenNr)
119:      END PufferEinfuegen;
120:
121:    BEGIN (* Eingabe *)
122:      ZeilenNr:=1; Zustand:=ZeichenLesen; ReadChar(Zeichen);
123:      WHILE Done DO
124:        CASE Zustand OF
125:          ZeichenLesen : CASE CAP(Zeichen) OF
126:                           'A'..'Z' : ZeichenZahl:=0;
127:                                      Puffer[ZeichenZahl]:=Zeichen;
128:                                      Zustand:=WortLesen
129:                         | '"'      : Zustand:=String1
130:                         | "'"      : Zustand:=String2
131:                         | "("      : ReadChar(Naechstes);
132:                                      IF Naechstes='*'
133:                                      THEN
134:                                        KommentarTiefe:=1;
135:                                        Zustand:=Kommentar
136:                                      END;
137:                                      PushBack(Naechstes)
138:                         END (* CASE *)
139:        | WortLesen    : CASE CAP(Zeichen) OF
140:                           'A'..'Z',
141:                           '0'..'9',
142:                           '^' : Anhaengen(Zeichen)
143:                         | '.' : ReadChar(Naechstes);
144:                                 IF (CAP(Naechstes)>='A') AND
145:                                    (CAP(Naechstes)<='Z')
```

```
146:                                    THEN
147:                                      Anhaengen('.');
148:                                      Anhaengen(Naechstes)
149:                                    ELSE
150:                                      PushBack(Naechstes);
151:                                      PufferEinfuegen;
152:                                      Zustand:=ZeichenLesen
153:                                    END
154:                            ELSE
155:                              IF Zeichen<>EOL
156:                              THEN PushBack(Zeichen)
157:                              END; (* IF *)
158:                              PufferEinfuegen;
159:                              Zustand:=ZeichenLesen
160:                            END (* CASE *)
161:          | String1       : IF Zeichen="'" THEN Zustand:=ZeichenLesen END
162:          | String2       : IF Zeichen='"' THEN Zustand:=ZeichenLesen END
163:          | Kommentar     : CASE Zeichen OF
164:                              '(' : ReadChar(Naechstes);
165:                                    IF Naechstes='*'
166:                                    THEN INC(KommentarTiefe)
167:                                    END;
168:                                    PushBack(Naechstes)
169:                             | '*' : ReadChar(Naechstes);
170:                                    IF Naechstes=')'
171:                                    THEN
172:                                      DEC(KommentarTiefe);
173:                                      IF KommentarTiefe=0
174:                                      THEN Zustand:=ZeichenLesen
175:                                      END
176:                                    END;
177:                                    PushBack(Naechstes)
178:                            END (* CASE *)
179:          END; (* CASE *)
180:          IF Zeichen=EOL THEN INC(ZeilenNr) END;
181:          ReadChar(Zeichen)
182:        END (* WHILE *)
183:      END Eingabe;
184:
185:    PROCEDURE Ausgabe(B : WortListe);
186:      VAR Anzahl, i, Nr : CARDINAL;
187:      BEGIN
188:        IF B<>NIL
189:        THEN WITH B^ DO
190:               Ausgabe(links);
191:               WITH Inhalt DO
192:                 WriteDynString(Wort);
193:                 IF DynStringLength(Wort)>WortBreite
194:                 THEN Anzahl:=(Spalten-DynStringLength(Wort))
195:                              DIV SpaltenProNr
196:                 ELSE Anzahl:=(Spalten-WortBreite) DIV SpaltenProNr
197:                 END; (* IF *)
198:                 FOR i:=DynStringLength(Wort)+1 TO WortBreite
199:                 DO Write('.')
200:                 END; (* FOR *)
201:                 WHILE NOT EmptyFIFO(Vorkommen) DO
202:                   IF Anzahl=0
203:                   THEN
204:                     Anzahl:=(Spalten-WortBreite) DIV SpaltenProNr;
205:                     WriteLn; FOR i:=1 TO WortBreite DO Write(' ') END
206:                   END; (* IF *)
```

```
207:                GetFromFIFO(Vorkommen,Nr);
208:                WriteCard(Nr,SpaltenProNr-1);
209:                IF NOT EmptyFIFO(Vorkommen) THEN Write(",") END;
210:                DEC(Anzahl)
211:              END; (* WHILE *)
212:              WriteLn
213:            END; (* WITH Inhalt *)
214:            Ausgabe(rechts)
215:          END (* WITH B^ *)
216:      END (* IF *)
217:    END Ausgabe;
218:
219:  BEGIN (* QuerverweisListe *)
220:    WriteString("Querverweisliste"); WriteLn;
221:    WriteString("----------------"); WriteLn;
222:    WriteLn;
223:    OpenInput("MOD");
224:    IF NOT Done THEN WriteString("Keine Datei!"); HALT END;
225:    Eingabe(Woerter);
226:    CloseInput;
227:    OpenOutput("");
228:    Ausgabe(Woerter);
229:    CloseOutput;
230:  END QuerverweisListe.
```

Aufgabe:

Die Prozedur «DynStringLess» aus unserem Bibliotheksmodul «DynStr» wurde so abgeändert, daß (für die alphabetische Reihenfolge) nicht zwischen Groß- und Kleinbuchstaben unterschieden wird. Wie schaut diese Änderung aus?

Anhang A

Lösungen der Aufgaben

Die angegebenen Lösungen zu den einzelnen Übungsaufgaben sind in den meisten Fällen sehr ausführlich. Bei komplexeren Übungen ist jedoch manchmal nur ein möglicher Lösungsweg skizziert worden.

Kapitel 2.4

1a: ☐ Verschiedene Schreibweise: LeeresProgramm und Leeresprogramm
☐ Das Modul wird mit einem Punkt abgeschlossen, nicht mit einem Semikolon.

1b: ☐ Der Modulname ist kein korrekter Bezeichner – der Unterstrich ist nicht erlaubt.
☐ Auf Modulname folgt ein Semikolon, kein Komma.
☐ Das Bibliotheksmodul heißt InOut, und nicht INOUT.
☐ In der Importliste werden die einzelnen Bezeichner durch Komma getrennt.
☐ 'Modula-2 mit Fehlern!" ist eine illegale String-Konstante.
☐ Zwischen «WriteString...» und «WriteLn» muß ein Semikolon stehen.
☐ Der Name am Modulende stimmt nicht mit dem Modulnamen überein.

2. "Symbol" – Das Symbol muß in der angegeben Form geschrieben werden.
   a|b – Entweder a oder b.
   [a] – a kann einmal vorkommen, muß aber nicht.
   {a} – a kann beliebig oft vorkommen.
   ::= – definitionsgemäß gleich.

3. String-Konstante::='"'Zeichenfolge'"' | "'"Zeichenfolge"'".
   Zeichenfolge::={Zeichen}.

Kapitel 3.3

1.
```
MODULE Differenz;
 FROM InOut IMPORT ReadCard, WriteCard, WriteLn, WriteString;
 VAR  A, B : CARDINAL;
 BEGIN
   WriteString("Bitte geben Sie zwei Zahlen A und B ein.");
   WriteLn;
   WriteString("A = "); ReadCard(A); WriteLn;
   WriteString("B = "); ReadCard(B); WriteLn;
   WriteString("A - B = "); WriteCard(A-B,1);
   WriteLn
 END Differenz.
```

2. Das Programm kann wie «Differenz» aufgebaut werden mit folgenden Unterschieden:

```
VAR A, B, C : CARDINAL; ...
WriteString("C = "); ReadCard(C); WriteLn; ...
WriteString("A*B*C = "); WriteCard(A*B*C,1); ...
```

Kapitel 3.4

1. Die Hauptarbeit liegt im Austausch von CARDINAL durch INTEGER. Entsprechend müssen die Ein- und Ausgabeanweisungen geändert werden.

Kapitel 3.6

1. Das Programm kommt mit einer einzigen CHAR-Variablen aus:

```
MODULE GrossUndKlein; (* Mit nur einer Variablen *)
  FROM InOut IMPORT Read, Write, WriteString, WriteLn;
  VAR Grossbuchstabe : CHAR;
  BEGIN
    WriteLn; WriteString("Bitte geben Sie einen Großbuchstaben ein: ");
    Read(Grossbuchstabe);
    WriteString(" - > "; Write(CHR(ORD(Grossbuchstabe)+32)); WriteLn
  END GrossUndKlein.
```

Obwohl das Programm wesentlich kürzer wird, nimmt die Klarheit ab!

2.
```
MODULE SchreibOktal;

FROM InOut IMPORT ReadCard, WriteCard, WriteString, WriteLn;

VAR Zahl, Stelle1, Stelle2, Stelle3 : CARDINAL;
```

```
BEGIN
  WriteString("Bitte geben Sie eine positive Zahl bis 127 ein: ");
  ReadCard(Zahl); WriteLn;
  WriteString("Die Oktaldarstellung dieser Zahl ist: ");
  Stelle1:=Zahl DIV 64;
  Stelle2:=(Zahl MOD 64) DIV 8;
  Stelle3:=(Zahl MOD 64) MOD 8;
  WriteCard(Stelle1,1); WriteCard(Stelle2,1); WriteCard(Stelle3,1);
  WriteLn
END SchreibOktal.
```

Kapitel 3.7.7

1. TRUE AND NOT (FALSE OR (TRUE AND NOT FALSE)) <–>
   TRUE AND NOT (FALSE OR (TRUE AND TRUE)) <–>
   TRUE AND NOT (FALSE OR TRUE) <–>
   TRUE AND NOT TRUE <–>
   TRUE AND FALSE <–>
   FALSE <–>

2a. x<=y

b. Zur Vereinfachung nehmen wir folgende Ersetzung vor:
   p := NOT(a<b) und
   q := (c>d).
   Die angegebene Formel hat nun folgende Gestalt:
   NOT (p AND q).
   Darauf kann die Regel von De Morgan angewandt werden:
   NOT p OR NOT q.
   Ersetzt man nun p wieder durch seine ursprüngliche Form, so fällt die doppelte Negation (NOT (NOT ..) wieder weg:
   (a<b) OR NOT q.
   Bei q kann die Negation in den Vergleichsoperator übernommen werden:
   (a<b) OR (c<=d).

c. NOT p OR q

Kapitel 3.8

1. CARDINAL-Zahl::=Dezimalziffer{Dezimalziffer}.
   Dezimalziffer::="0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

   Nimmt man auch die Oktal- und Hexadezimaldarstellung mit auf, so lautet die Definition folgendermaßen:

   CARDINAL-Zahl::=Oktalzahl | Dezimalzahl | Hexadezimalzahl.

Oktalzahl::=Oktalziffer{Oktalziffer}"B".
Oktalziffer::="0"|"1"|"2"|"3"|"4"|"5"|"6"|"7".
Dezimalzahl::=Dezimalziffer{Dezimalziffer}.
Dezimalziffer::=Oktalziffer|"8"|"9".
Hexadezimalzahl::=Dezimalziffer{Hexziffer}"H".
Hexziffer::=Dezimalziffer|"A"|"B"|"C"|"D"|"E"|"F".

INTEGER-Zahl::=["+"|"-"]CARDINAL-Zahl.

2. Das Programm kann direkt aus «Volumen» gewonnen werden, indem die Variable «Hoehe» weggelassen und «Volumen» durch «Fläche» ersetzt wird.

Kapitel 4.1

1. Mehrfach verschachtelte IF-Anweisungen werden sehr schnell unübersichtlich. Zudem erhöht sich der Schreibaufwand, da jede IF-Anweisung ihr eigenes «END» benötigt.

2.
```
MODULE GrossKlein;
FROM InOut IMPORT Read, Write;
VAR Eingabe : CHAR;
BEGIN
  Read(Eingabe);
  IF (Eingabe>="A") AND (Eingabe<="Z")
  THEN Write(CHR(ORD(Eingabe)+ORD("a")-ORD("A")))
  ELSIF (Eingabe>="a") AND (Eingabe<="z")
  THEN Write(CAP(Eingabe))
  ELSE Write(Eingabe)
  END (* IF *)
END GrossKlein.
```

Kapitel 4.2

1.
```
MODULE DezimalbruchAusgabe;
FROM InOut IMPORT WriteLn, WriteString, WriteCard, ReadCard, Write;
VAR A, B, Rest, Nachkommastellen : CARDINAL;
BEGIN
  WriteString("Bitte geben Sie zwei natürliche Zahlen A und B ein.");
  WriteLn;
  WriteString("A = "); ReadCard(A); WriteLn;
  WriteString("B = "); ReadCard(B); WriteLn;
  WriteString("A/B = "); WriteCard(A DIV B,1); Write(".");
  Nachkommastellen:=0; Rest:=A MOD B;
  REPEAT
    WriteCard((10*Rest) DIV B,1);
    Rest:=(10*Rest) MOD B;
    INC(NachKommastellen)
  UNTIL Nachkommastellen=10;
  WriteLn
END DezimalbruchAusgabe.
```

2.
```
MODULE Durchschnitt;
FROM InOut IMPORT WriteLn, WriteString, WriteCard, ReadCard, Write;
VAR Zahl, Summe, Anzahl, Rest : CARDINAL;
BEGIN
  WriteString("Durchschnittswert-Berechnung"); WriteLn;
  WriteString("Geben Sie eine Folge von natürlichen Zahlen ein.");
  WriteLn;
  WriteString("Ende = 0"); WriteLn;
  Anzahl:=0; Summe:=0;
  REPEAT
    ReadCard(Zahl);
    INC(Summe,Zahl); (* Oder Summe:=Summe+Zahl *)
    IF Zahl<>0 THEN INC(Anzahl) END
  UNTIL Zahl=0;
  WriteString("Anzahl der eingebenen Zahlen: ");
  WriteCard(Anzahl,1); WriteLn;
  IF Anzahl>0
  THEN
    WriteString("Der Durchsnittswert beträgt: ");
    WriteCard(Summe DIV Anzahl,1);
    Write(".");
    Rest:=Summe MOD Anzahl;
    WriteCard((10*Rest) DIV Anzahl,1);
    Rest:=(10*Rest) MOD Anzahl;
    WriteCard((10*Rest) DIV Anzahl,1);
    WriteLn
  END (* IF *)
END Durchschnitt.
```

Kapitel 4.3

1.
```
MODULE Kopieren; (* Version mit REPEAT-Anweisung *)
  FROM InOut IMPORT Read, Done, OpenInput, CloseInput, OpenOutput,
                    CloseOutput, Write, WriteLn, WriteString;
  VAR Zeichen : CHAR;
  BEGIN
   WriteLn;
   WriteString("Kopieren von Text-Dateien"); WriteLn;
   WriteString("-------------------------"); WriteLn;
   WriteLn;
   OpenInput("MOD"); WriteLn;
   IF NOT Done
   THEN WriteString("Datei existiert nicht!"; WriteLn; HALT
   END; (* IF *)
   OpenOutput("MOD");
   Read(Zeichen);
   IF Done
   THEN REPEAT
          Write(Zeichen);
          Read(Zeichen)
        UNTIL NOT Done
   END; (* IF *)
   CloseInput; CloseOutput;
   WriteString("Kopie erstellt."); WriteLn;
  END Kopieren.
```

2.
```
MODULE Count;
  FROM InOut IMPORT OpenInput, CloseInput, Read, WriteLn,
                    WriteCard, WriteString;
  VAR Zeichen : CHAR;
      GrossBuchstabenZahl, KleinBuchstabenZahl : CARDINAL;
  BEGIN
   WriteLn;
   WriteString("Anzahl der Buchstaben in einem Text"); WriteLn;
   WriteString("-----------------------------------"); WriteLn;
   WriteLn;
   OpenInput("MOD"); WriteLn;
   IF NOT Done
   THEN WriteString("Datei existiert nicht!"; WriteLn; HALT
   END; (* IF *)
   GrossBuchstabenZahl:=0; KleinBuchstabenZahl:=0;
   Read(Zeichen);
   WHILE Done DO
     IF (Zeichen>="A") AND (Zeichen<="Z")
     THEN GrossBuchstabenZahl:=GrossBuchstabenZahl+1
     ELSIF (Zeichen>="a") AND (Zeichen<="z")
     THEN KleinBuchstabenZahl:=KleinBuchstabenZahl+1
     END; (* IF *)
     Read(Zeichen)
   END; (* WHILE *)
   CloseInput;
   WriteString("Analyse beendet."); WriteLn;
   WriteString("Es wurden ");
   WriteCard(GrossBuchstabenZahl,1);
   WriteString(" Großbuchstaben und ");
   WriteCard(KleinBuchstabenZahl,1);
   WriteString(" Kleinbuchstaben gefunden.");
   WriteLn
  END Count.
```

3. Es reicht die Änderung von 'Write(Zeichen)' in 'Write (CAP(Zeichen))'.

Kapitel 4.4

1. Es genügt die Änderung der WHILE-Anweisung:

```
LOOP
  IF Eingabe=0 THEN EXIT END;
  Fakultaet:=Fakultaet*Eingabe;
  Eingabe:=Eingabe-1
END; (* LOOP *)
```

2. Genauso kann auch im Kopierprogramm die WHILE-Anweisung ersetzt werden:

```
LOOP
  IF NOT Done THEN EXIT END;
  Write(Zeichen);
  Read(Zeichen)
END; (* LOOP *)
```

Kapitel 4.5

1.
```
MODULE SummeBisX;
FROM InOut IMPORT ReadCard, WriteCard, WriteLn, WriteString;
VAR Summe, X, i : CARDINAL;
BEGIN
  WriteString("Geben Sie eine natürliche Zahl ein: ");
  ReadCard(X); WriteLn;
  WriteString("Die Summe aller natürlichen Zahlen bis ");
  WriteCard(X,1);
  WriteString(" beträgt: ");
  Summe:=0; FOR i:=1 TO X DO INC(Summe,i) END;
  WriteCard(Summe,1); WriteLn
END SummeBisX.
```

2. Hier nur die Hauptschleife:

```
FOR i:=MAX(CARDINAL) TO 0 BY -1 DO
  IF FLOAT(i)>=Eingabe THEN Minimum:=i END
END; (* FOR *)
```

3. Die REPEAT-Anweisung wird folgendermaßen ersetzt:

```
FOR Nachkommastellen:=1 TO 10 DO
  WriteCard((10*Rest) DIV B,1);
  Rest:=(10*Rest) MOD B
END; (* FOR *)
```

Kapitel 4.6

1. Der Weg wird in der Lösung von Aufgabe 2, Kapitel 6 beschrieben.

2.
```
MODULE ZeichenZaehlen;
FROM InOut IMPORT OpenInput, Read, Done, WriteString,
                  WriteLn, WriteCard;
VAR Grossbuchstaben, Kleinbuchstaben, Sonderzeichen : CARDINAL;
    Zeichen : CHAR;
BEGIN
  WriteString("Textanalyse"); WriteLn;
  WriteString("-----------"); WriteLn;
  Grossbuchstaben:=0; Kleinbuchstaben:=0; Sonderzeichen:=0;
  OpenInput("");
  WHILE Done DO
    Read(Zeichen);
    CASE Zeichen OF
      "A".."Z" : INC(Grossbuchstaben);
    | "a".."z" : INC(Kleinbuchstaben);
    ELSE INC(Sonderzeichen)
    END (* CASE *)
  END; (* WHILE *)
  CloseInput;
  WriteString("Der Text besteht aus"); WriteLn;
  WriteCard(Grossbuchstaben,1);
  WriteString(" Großbuchstaben,"); WriteLn;
  WriteCard(Kleinbuchstaben,1);
```

```
    WriteString(" Kleinbuchstaben und "); WriteLn;
    WriteCard(Sonderzeichen,1);
    WriteString(" Sonderzeichen.")
END ZeichenZaehlen.
```

Kapitel 5.3

1.
```
MODULE M;
VAR X,Y : CARDINAL;
    Z   : REAL;

PROCEDURE P;
  VAR A,B,X,Y : INTEGER;
      Z : CARDINAL;

  PROCEDURE P1;
    VAR A,B : CHAR
    BEGIN
      (* Hier bekannte Bezeichner:
          A, B          von P1
          X, Y, Z, P1   von P
          P             von M *)
    END P1;

  PROCEDURE P2;
    VAR P : CARDINAL;
        X : REAL;

    PROCEDURE P;
     (* Dieser Prozedurname ist nicht zuläßig, da P bereits
        durch die lokale CARDINAL-Variable belegt ist! *)
      VAR A,B : REAL;
      BEGIN
      END P;

    BEGIN (* P2 *)
      (* Hier bekannte Bezeichner
          P, X                 von P2
          A, B, Y, Z, P1, P2 von P
          die Prozedur P von M ist hier nicht sichtbar! *)
    END P2;

  BEGIN
    (* Hier bekannte Bezeichner:
        A, B, X, Y, Z, P1, P2  von P
        P                      von M *)
  END P;

BEGIN
    (* Hier bekannte Bezeichner:
        X, Y, Z, P1, P2 von M
END M.
```

2.
```
 2
12
 1
 2
12
```

Kapitel 5.4

1. `PROCEDURE xyz(VAR A,B : CARDINAL; C : CARDINAL); ...`

   `xyz(n,m,3-9)` **falsch**, da der Ausdruck 3-9 keinen CARDINAL-Wert liefert.
   `xyz(n,n,m)` **korrekt**
   `xyz(n,1000 DIV m,m)` **falsch**, da Ausdruck an zweiter Stelle.
   `xyz(n,m,2*m-n+ORD("A"))` **korrekt**, aber u.U. Laufzeitfehler, wenn der Ausdruck negativ wird.

2. Ausgabeprozeduren: Wertparameter
   Eingabeprozeduren: Referenzparameter

Kapitel 5.5

Das Problem ist am einfachsten zu lösen, wenn man eine zusätzliche Variable deklariert, die anzeigt, ob bereits eine Ziffer gelesen wurde. Hat diese boolesche Variable den Wert FALSE, so werden Leerzeichen ignoriert, ansonsten führen sie zu einem Fehler. Ist sie auch am Ende noch FALSE, so wurde eine Leerzeile eingegeben.

```
PROCEDURE LiesCard(VAR Zahl, FehlerCode : CARDINAL);
  VAR Zeichen : CHAR;
      ZifferGelesen : BOOLEAN;
  BEGIN
    Zahl:=0; FehlerCode:=0; ZifferGelesen:=FALSE;
    REPEAT
      Read(Zeichen);
      CASE Zeichen OF
         " " : IF ZifferGelesen THEN FehlerCode:=1 END;
       | EOL : (* nichts *)
       | '0'..'9' : IF Zahl<=(MAX(CARDINAL)-(ORD(Zeichen)-ORD('0'))) DIV 10
                    THEN Zahl:=10*Zahl+(ORD(Zeichen)-ORD('0')
                    ELSE FehlerCode:=2
                    END;
                    ZifferGelesen:=TRUE
      ELSE FehlerCode:=1
      END (* CASE *)
    UNTIL (Zeichen=EOL) OR (FehlerCode>0);
    IF NOT ZifferGelesen THEN FehlerCode:=3 END
  END LiesCard;
```

2. Bei der Eingabe von INTEGER-Zahlen ist es günstig, in einer weiteren Variablen ein eventuell eingegebenes Vorzeichen zu speichern. Die boolesche Variable «VorzeichenGelesen» zeigt diesen Sachverhalt an.

```
PROCEDURE LiesInt(VAR Zahl : INTEGER; VAR FehlerCode : CARDINAL);
  VAR Zeichen : CHAR;
      ZifferGelesen, VorzeichenGelesen : BOOLEAN;
      Vorzeichen : INTEGER;
```

```
BEGIN
  Zahl:=0; FehlerCode:=0; ZifferGelesen:=FALSE;
  Vorzeichen:=+1; VorzeichenGelesen:=FALSE;
  REPEAT
    Read(Zeichen);
    CASE Zeichen OF
       " " : IF ZifferGelesen THEN FehlerCode:=1 END;
     | "+" : IF VorzeichenGelesen OR ZifferGelesen
             THEN FehlerCode:=1
             ELSE VorzeichenGelesen:=TRUE
             END;
     | "-" : IF VorzeichenGelesen OR ZifferGelesen
             THEN FehlerCode:=1
             ELSE Vorzeichen:=-1; VorzeichenGelesen:=TRUE
             END
     | EOL : (* nichts *)
     | '0'..'9' : IF Zahl<=(MAX(INTEGER) DIV 10
                  THEN Zahl:=10*Zahl+(ORD(Zeichen)-ORD('0')
                  ELSE FehlerCode:=2
                  END;
                  ZifferGelesen:=TRUE
    ELSE FehlerCode:=1
    END (* CASE *)
  UNTIL (Zeichen=EOL) OR (FehlerCode>0);
  IF NOT ZifferGelesen THEN FehlerCode:=3 END
END LiesInt;
```

"+" und "-" können auch unter einem CASE-Label zusammengefaßt werden:

```
  | "+","-" : IF VorzeichenGelesen OR ZifferGelesen
              THEN FehlerCode:=1
              ELSE
                VorzeichenGelesen:=TRUE;
                IF Zeichen="-" THEN Vorzeichen:=-1 END
              END;
```

Kapitel 5.6

1.
```
PROCEDURE CARD(Ziffer : CHAR) : CARDINAL;
  BEGIN
    RETURN ORD(Ziffer)-ORD("0")
  END CARD;
```

2.
```
PROCEDURE IstGerade(Zahl : CARDINAL) : BOOLEAN;
  BEGIN
    RETURN NOT ODD(Zahl)
    (* oder Zahl MOD 2 = 0 *)
  END IstGerade;
```

3. Das einfachste Verfahren besteht darin, erst zu prüfen, ob es sich bei der zu testenden Zahl um 2 handelt (was ja eine Primzahl ist) oder ob sie gerade ist. Ansonsten wird die Zahl durch sämtliche ungeraden Zahlen geteilt. Ist sie durch eine solche ohne Rest teilbar, so handelt es sich um keine Primzahl. Die größte ungerade Zahl, mit der der Test durchgeführt wer-

den muß, ist dann erreicht, wenn das Produkt dieser Zahl mit sich selbst den Probanden überschreitet. Sind alle Tests negativ verlaufen, so handelt es sich um eine Primzahl.

```
PROCEDURE Primzahl(Zahl : CARDINAL) : BOOLEAN;
  VAR TestZahl : CARDINAL;
  BEGIN
    IF Zahl=2 THEN RETURN TRUE
    ELSIF NOT ODD(Zahl) THEN RETURN FALSE
    ELSE
      TestZahl:=3;
      WHILE TestZahl*TestZahl<=Zahl DO
        IF Zahl MOD TestZahl = 0 THEN RETURN FALSE END;
        TestZahl:=TestZahl+2 (* oder INC(TestZahl,2) *)
      END; (* WHILE *)
      RETURN TRUE
    END (* IF *)
  END Primzahl;
```

4.
```
PROCEDURE FRAC(Zahl : REAL):REAL;
  BEGIN
    RETURN Zahl-INT(Zahl)
  END FRAC;
```

Kapitel 6.3

2. Zeichenketten dürfen, wenn sie nicht in Kommentaren stehen, komplett ausgegeben werden. So kann das Problem durch einen weiteren ELSIF-Zweig gelöst werden:

   ... ELSIF (Zeichen="'") AND (Kommentartiefe=0)
   THEN REPEAT Write(Zeichen); Read(Zeichen) UNTIL Zeichen="'"

Der Fall, daß Zeichenketten in einfache Hochkommas eingeschlossen sind, kann mit derselben Methode behandelt werden.

Übrigens birgt das Programm noch eine weitere Fehlerquelle, wenn die Eingabe kein korrektes Modula-2-Programm darstellt. Endet der zu analysierende Text nämlich mit (oder *, so wird durch «read(Naechstes)» über das Ende der Datei hinaus gelesen, was unter Umständen zu einem Programmabbruch führt.

Dieser Fehler kann am einfachsten dadurch behoben werden, daß auf das folgende Zeichen nur dann zugegriffen wird, wenn Done den Wert «TRUE» hat:

IF (Zeichen='(') AND Done ...
IF (Zeichen='(*) AND Done ...

Allerdings muß dann sicherheitshalber auch vor dem zweiten «ReadChar» eine entsprechende Abfrage stattfinden.

Eine andere Möglichkeit besteht darin, bereits die Prozedur «ReadChar» im Modul «GepuffertesLesen» so zu gestalten, daß nicht über das Ende der Datei hinaus gelesen werden kann. Die Abfrage könnte hier beispielsweise so aussehen:

```
IF (Puffer=0C) AND Done ...
```

Kapitel 7.1.1

Der IMPORT-Teil muß um folgende Liste erweitert werden:

```
FORM Terminal IMPORT BusyRead;
```

Dann muß eine CHAR-Variable deklariert werden:

```
VAR Eingabe : CHAR;
```

Innerhalb der LOOP-Anweisung müssen folgende Anweisungen eingefügt werden:

```
BusyRead(Eingabe); IF Eingabe<>0C THEN HALT END;
```

Eine andere Möglichkeit besteht darin, die LOOP- durch eine REPEAT-Anweisung so zu ersetzen:

```
REPEAT
...
BusyRead(Eingabe)
UNTIL Eingabe<>0C
```

Kapitel 7.1.2

1. Das Programm kann nahezu vollständig aus «StringExtrakt» gewonnen werden. Der Hauptunterschied liegt darin, daß jedes gelesene Zeichen ausgegeben werden muß, im Zustand «Zeichenlesen» in Großbuchstaben, ansonsten unverändert.

```
MODULE KopierenMitUmwandlung;

  FROM InOut IMPORT OpenInput, CloseInput, Done, Read, Write, WriteString,
                    WriteLn, OpenOutput, CloseOutput;

  TYPE AutomatenZustand = (ZeichenLesen, String1Lesen, String2Lesen);

  VAR  Zustand : AutomatenZustand;
       Zeichen : CHAR;

  BEGIN
    WriteLn;
    WriteString("Umwandlung eines Modula-2-Programmtextes in Versalien");
    WriteLn;
    WriteString("Stringkonstanten bleiben unberührt");
    WriteLn; WriteLn;
    OpenInput("MOD");
    IF NOT Done THEN WriteString("Kein File!"); WriteLn; HALT END;
    OpenOutput("");
    Zustand:=ZeichenLesen;
    Read(Zeichen);
    WHILE Done DO
      CASE Zustand OF
        ZeichenLesen : Write(CAP(Zeichen));
                       IF Zeichen="'"
                       THEN Zustand:=String1Lesen
                       ELSIF Zeichen='"'
                       THEN Zustand:=String2Lesen
                       END
      | String1Lesen : Write(Zeichen);
                       IF Zeichen="'"
                       THEN Zustand:=ZeichenLesen
                       END
      | String2Lesen : Write(Zeichen);
                       IF Zeichen='"'
                       THEN Zustand:=ZeichenLesen
                       END
      END; (* CASE *)
      Read(Zeichen)
    END; (* WHILE *)
    CloseInput;
    CloseOutput
  END KopierenMitUmwandlung.
```

2. Ohne diese Abfrage würde ein leerer Text das Programm zusammenbrechen lassen: Division durch 0!
3. Der Automat benötigt die Zustände «Vorzeichen» und «Ziffernfolge». Da nicht bis zum Ende der Datei gelesen wird, benötigt man noch einen speziellen Zustand («Ende»), der das Ende eine Zahl anzeigt. Startzustand ist «Vorzeichen»:

Alter Zustand	Zeichen	Aktion	neuer Zustand
Vorzeichen	'+','–'	speichern Zahl mit 0 initialisieren	Ziffernfolge
	'0'..'9'	Vorzeichen:=+1 Zahl entsprechend Ziffer initialisieren	Ziffernfolge
	sonst	Fehler	Ende
Ziffernfolge	'0'..'9'	Zahl weiterführen	Ziffernfolge
	Begrenzer	–	Ende
	sonst	Fehler	Ende

Begrenzer könnten beispielsweise sein: ' ', EOL.

Kapitel 7.2

Die Compilerschalter sind nicht genormt, jeder Anbieter geht hier seine eigenen Wege. Aus diesem Grund müssen Sie zur Lösung Ihr Handbuch konsultieren.

Kapitel 7.4.1

F	HIGH(F)	
Feld4	5	(6 Farben)
Feld5	1	(2 Wahrheitswerte)
Feld6	10	(11 Zahlen)

Kapitel 7.4.2

1.
```
MODULE StringprozedurenTest;

  FORM InOut IMPORT ReadString, WriteString, WriteLn, WriteCard;

  VAR String1, String2, String3 : ARRAY[1..80] OF CHAR;

  (* hier die Prozeduren einfügen *)

  BEGIN
    WriteString("Bitte geben sie zwei Wörter ein:"); WriteLn;
    ReadString(String1); WriteLn;
    ReadString(String2); WriteLn;
    WriteString("Das erste Wort besteht aus ");
    WriteCard(StringLaenge(String1),1);
    WriteString(" Zeichen."); WriteLn;
    WriteString("Das zweite Wort besteht aus ");
    WriteCard(StringLaenge(String2),1);
    WriteString(" Zeichen."); WriteLn;
```

```
    WriteString("Zusammengesetzt ergibt sich:"); WriteLn;
    Concat(String1, String2, String3);
    WriteString(String3); WriteLn;
    WriteString("Das zweite Wort ist im ersten ");
    IF Pos(String2, String1)>0
    THEN
      WriteString("an "); WriteCard(Pos(String2, String1),1);
      WriteString("-ter Stelle ")
    ELSE WriteString("nicht ")
    END;
    WriteString("enthalten.") WriteLn
  END StringprozedurenTest.
```

2. Die Anzahl der benötigten Leerzeichen ist n-StringLaenge(String). Da der String rechtsbündig ausgedruckt werden soll, müssen die Leerzeichen vor dem String ausgegeben werden.

```
PROCEDURE Print(String : ARRAY OF CHAR; n : CARDINAL);
  VAR i : CARDINAL;
  BEGIN
    FOR i:=1 TO n-StringLaenge(String) DO Write(' ') END;
    WriteString(String)
  END Print;
```

Möglich ist auch: FOR i:=StringLaenge(String)+1 TO n DO ...

3.
```
DEFINITION MODULE Strings;

  PROCEDURE StringLaenge(Zeile : ARRAY OF CHAR) : CARDINAL;
  (* Liefert die Länge einer Zeichenkette *)

  PROCEDURE Concat(S1,S2 : ARRAY OF CHAR; VAR S : ARRAY OF CHAR);
    (* Setzt den String S aus den beiden Teilstrings S1 und S2
       zusammen *)

  PROCEDURE Pos(Suchstring, String : ARRAY OF CHAR) : CARDINAL;
    (* Gibt das erste Auftreten des Suchstrings in einem String
       zurück. 0 bedeutet, daß der Suchstring nicht enthalten ist *)

  END Strings.

IMPLEMENTATION MODULE Strings;
```

Hier einfach die drei Prozeduren einkopieren

```
  END Strings.
```

Kapitel 7.4.3

1. In diesem Fall ist es am einfachsten, die Zahl von hinten abzubauen. Der Algorithmus lautet:

   Wiederhole
   die nächste Ziffer (von hinten) ergibt sich aus Zahl MOD Basis
   Zahl ergibt sich aus DIV Basis
   bis die Zahl den Wert 0 erreicht hat.

   Da die fortgesetzte Teilung nur sehr wenige Durchläufe benötigt, verwenden wir denselben Algorithmus zur Bestimmung der benötigten Anzahl von Ziffern.

```
PROCEDURE CardToString(VAR String : ARRAY OF CHAR; Zahl, Basis :CARDINAL);
  VAR AnzahlZiffern, Ziffer, temp : CARDINAL;
  BEGIN
    AnzahlZiffern:=0; temp:=Zahl;
    REPEAT temp:=temp DIV Basis; INC(AnzahlZiffern) UNTIL temp=0;
    IF AnzahlZiffern<HIGH(String) THEN RETURN END; (* String zu kurz! *)
    IF Basis=2 THEN String[AnzahlZiffern]:='B'
    ELSIF Basis=8 THEN String[AnzahlZiffern]:='O'
    ELSIF Basis=10 THEN String[AnzahlZiffern]:='D'
    ELSIF Basis=16 THEN String[AnzahlZiffern]:='H'
    END;
    IF HIGH(String)>AnzahlZiffern THEN String[AnzahlZiffern+1]:=0C END;
    REPEAT
      DEC(AnzahlZiffern);
      Ziffer:=Zahl MOD Basis; Zahl:=Zahl DIV Basis;
      IF Ziffer<10
      THEN String[AnzahlZiffern]:=CHR(Ziffer+ORD('0'))
      ELSE String[AnzahlZiffern]:=CHR(Ziffer+10+ORD('A'))
      END
    UNTIL Zahl=0
  END CardToString;
```

2. Die Prozedur «LiesCard», die keine Fehleingaben zuläßt, besteht in der Hauptsache nur aus einer Schleife, die solange durchlaufen wird, bis ein eingegebener String fehlerfrei umgewandelt werden kann:

```
PROCEDURE LiesCard(VAR Zahl : CARDINAL);
  VAR Eingabe : ARRAY[1..80] OF CHAR;
      Fehler : CARDINAL;
  BEGIN
    REPEAT
      ReadString(Eingabe);
      StringToCard(Eingabe, Zahl, Fehler);
      IF Fehler>0
      THEN WriteString(" ???")
      END;
      WriteLn
    UNTIL Fehler=0
  END LiesCard;
```

Kapitel 7.4.4

Die Lösung finden Sie in der Prozedur «vorhanden» (lokal zu «ReserviertesWort») im Programm «QuelltextLister».

Kapitel 7.4.5

Gehen Sie folgendermaßen vor:

a) Schreiben Sie die Prozedur «KursivAn», die den Drucker auf kursive Schrift umschaltet.
b) Schreiben Sie das Gegenstück hierzu («KursivAus»).
c) Bei der Zustandsänderung «ZeichenLesen» –> «Kommentar» fügen Sie «KursivAn» ein.
d) Bei der Zustandsänderung «Kommentar» –> «ZeichenLesen» fügen Sie «KursivAus» ein.

Kapitel 7.5.3

Obwohl der boolesche Ausdruck sehr kompliziert aussieht, kann der Funktionswert direkt angegeben werden:

```
PROCEDURE FrueherAls(Datum1, Datum2 : Datum) : BOOLEAN;
  BEGIN
    RETURN (Datum1.Jahr<Datum2.Jahr) OR
           ((Datum1.Jahr=Datum2.Jahr) AND (Datum1.Monat<Datum2.Monat)) OR
           ((Datum1.Jahr=Datum2.Jahr) AND (Datum1.Monat=Datum2.Monat)
            AND (Datum1.Tag<Datum2.Tag))
  END FrueherAls;
```

Ihre Lösung könnte eventuell auch so aussehen:

```
IF Datum1.Jahr<Datum2.Jahr THEN RETURN TRUE
ELSIF (Datum1.Jahr=Datum2.Jahr) AND (Datum1.Monat<Datum2.Monat)
THEN RETURN TRUE
ELSIF ...
```

Auf raffinierte Weise löst folgender Ausdruck das Problem:

```
RETURN (Datum1.Jahr<Datum2.Jahr) OR
       ((Datum1.Jahr=Datum2.Jahr) AND
        (Datum1.Monat*31+Datum1.Tag<Datum2.Monat*31+Datum2.Tag))
```

Die nochmalige Komprimierung auf

```
RETURN Datum1.Jahr*366+Datum1.Monat*31+Datum1.Tag<
       Datum2.Jahr*366+Datum2.Monat*31+Datum2.Tag
```

scheitert an einem CARDINAL-Überlauf.

Kapitel 7.5.4

1. Größter Bruch: MAX[NatuerlicheZahl]/1
   Kleinster positiver Bruch größer 0: 1/MAX[NatuerlicheZahl]
2. Bei der Funktion «BruchKleiner» müssen wiederum die Vorzeichen getrennt betrachtet werden. Ergebnis ist der Vergleich

$$a/b < c/d$$

```
PROCEDURE BruchKleiner(X, Y : Bruch) : BOOLEAN;
  BEGIN
    IF (X.Vorzeichen=-1) AND (Y.Vorzeichen=+1)
    THEN RETURN TRUE
    ELSIF (X.Vorzeichen=+1) AND (Y.Vorzeichen=-1)
    THEN RETURN FALSE
    ELSIF (X.Vorzeichen=-1) AND (Y.Vorzeichen=-1)
    THEN RETURN X.Zaehler*Y.Nenner>Y.Zaehler*X.Nenner
    ELSE RETURN X.Zaehler*Y.Nenner<Y.Zaehler*X.Nenner
    END
  END BruchKleiner;
```

Kapitel 7.5.5

1. Der gewünschte Datentyp hat folgende Struktur:

```
TYPE Adresse = RECORD
                 Name, Vorname : ARRAY[1..20] OF CHAR;
                 Strasse : ARRAY[1..40] OF CHAR;
                 PLZ : [1000..9000];
                 Ort : ARRAY[1..30] OF CHAR;
                 Telefon : ARRAY[1..12] OF CHAR
               END;
```

   Für die Ein- und Ausgabe bieten sich folgende Prozeduren an:

```
PROCEDURE LiesAdresse(VAR Eingabe : Adresse);
  BEGIN
    WITH Eingabe DO
      WriteString("Name: "); ReadString(Name); WriteLn;
      WriteString("Vorname: "); ReadString(Vorname); WriteLn;
      WriteString("Straße: "); ReadString(Strasse); WriteLn;
      WriteString("Postleitzahl: "); ReadCard(PLZ); WriteLn;
      (* kritisch !!! Besser in eine lokale Variable lesen und den
                      zulässigen Bereich selbst überprüfen *)
      WriteString("Ort: "); ReadString(Ort); WriteLn;
      WriteString("Telefon: "); ReadString(Telefon); WriteLn
    END (* WITH *)
  END LiesAdresse;

PROCEDURE SchreibAdresse(Ausgabe : Adresse);
  BEGIN
    WITH Ausgabe DO
    wie LiesAdresse, anstelle von Read... immer Write...,
    bei WriteCard Formatierung nicht vergessen!
  END SchreibAdresse;
```

```
PROCEDURE SchreibNamen(Ausgabe : Adresse);
  (* Gibt Name, Vorname ..... Telefon aus *)
  VAR i : CARDINAL;
  BEGIN
    WITH Ausgabe DO
      WriteString(Name); WriteString(", ");
      WriteString(Vorname);
      FOR i:=StringLaenge(Name)+StringLaenge(Vorname)+3 TO 60 DO
        Write('.')
      END; (* FOR *)
      WriteString(Telefon); WriteLn
    END (* WITH *)
  END SchreibNamen;
```

2. Das Programm ist prinzipiell genauso aufgebaut wie «Zahlen-Felder». Die Steuerung des Programms erfolgt hier über eine Menüauswahl, die als Funktionsprozedur ausgeführt wird und ein Resultat eines selbstdefinierten Aufzählungstyps «Menue-Punkt» liefert:

```
TYPE MenuePunkt = (Eingeben, AdressenAusgeben, TelefonListe,
                   Suchen, Beenden);

  PROCEDURE MenueAuswahl() : MenuePunkt;
    VAR Antwort : CHAR;
    BEGIN
      WriteString("E)ingeben, A)dressen, T)elefonliste, S)uchen, B)eenden ? ");
      REPEAT
        BusyRead(Antwort); Antwort:=CAP(Antwort)
      UNTIL (Antwort='E') OR (Antwort='A') OR (Antwort='T') OR
            (Antwort='S') OR (Antwort='B');
      Write(Antwort);
      CASE Antwort OF
         'E' : RETURN Eingeben
       | 'A' : RETURN AdressenAusgeben
       | 'T' : RETURN TelefonListe
       | 'S' : RETURN Suchen
       | 'B' : RETURN Beenden
      END (* CASE *)
    END MenueAuswahl;
```

Dadurch, daß «Beenden» als eigener Menüpunkt aufgenommen wurde, kann das Programm in einer Endlosschleife laufen:

```
CONST MaxAdressen = 100;

VAR AdressenListe : ARRAY[1..MaxAdressen] OF Adresse;
    i, FeldGroesse : [1..MaxAdressen];

LOOP
  CASE MenueAuswahl() OF
    Eingeben          : IF FeldGroesse<MaxAdressen
                        THEN
                          INC(FeldGroesse);
                          LiesAdresse(AdressenListe[FeldGroesse]);
                          Sortiere(AdressenListe,1,FeldGroesse)
```

```
                        ELSE WriteString("Liste ist voll!"); WriteLn
                        END
  | AdressenAusgeben : FOR i:=1 TO FeldGroesse DO
                          SchreibAdresse(AdressenListe[i]);
                          WriteLn
                        END
  | TelefonListe     : FOR i:=1 TO FeldGroesse DO
                          SchreibNamen(AdressenListe[i])
                        END
  ...
END;
```

3. Die Vorgehensweise entspricht exakt der beim Typ «Datum».

Kapitel 8.8

1. Ein Vorfahre ist entweder ein Vater oder eine Mutter, oder ein Vorfahre eines Vorfahren.

2.
```
PROCEDURE vertausche(VAR X,Y : CARDINAL);
  VAR t : CARDINAL;
  BEGIN
    t:=X; X:=Y; Y:=t
  END vertausche;
```

4. Hier der Kern des Programms:

```
FROM Random IMPORT RandomCard;
VAR Testfeld : ARRAY[0.999] OF CARDINAL;
    i : CARDINAL;
BEGIN
  FOR i:=0 TO 999 DO TestFeld[i]:=RandomCard(1000) END;
  Sortiere(TestFeld,0,999);
  FOR i:=0 TO 999 DO WriteCard(TestFeld[i],10); WriteLn END
END ...
```

5. Bei der Testreihe zeigt sich, daß die Sortierzeit annähernd linear ist, d. h. ein doppelt so großes Feld wird auch in der doppelten Zeit sortiert.

6. Anstelle des CARDINAL-Vergleichs muß nur der entsprechende Größenvergleich für Zeichenketten gesetzt werden. Das Feld selbst hat folgende Struktur:

   TYPE STRING = ARRAY[0..80] OF CHAR;
   VAR TestFeld : ARRAY[0..999] OF STRING;

   Ein zufällige Belegung kann auf folgende Weise realisiert werden:

```
VAR i,j, Laenge : CARDINAL;
...
FOR i:=0 TO 999 DO
  Laenge:=RandomCard(80);
  FOR j:=0 TO Laenge DO
    TestFeld[i][j]:=CHR(65+RandomCard(26)) (* ergibt Zeichen von "A" bis "Z" *)
  END; (* FOR j *)
  TestFeld[i][Laenge+1]:=0C
END (* FOR i *)
```

Da die Zuweisung auch für beliebige Felder gilt, kann die Prozedur «vertausche» wie oben gestaltet werden:

```
PROCEDURE vertausche(VAR X,Y : STRING);
  VAR t : STRING;
  BEGIN
    t:=X; X:=Y; Y:=t
  END vertausche;
```

7. Zu diesem Zweck machen wir am einfachsten aus den Prozeduren «CardinalAusdruck», «Term», «Faktor» und «CardinalZahl» Funktionsprozeduren vom Typ CARDINAL. Zusätzlich muß in jeder dieser Prozeduren eine lokale Variable für Zwischenergebnisse bereitgestellt werden.

```
PROCEDURE CardinalAusdruck() : CARDINAL;
  VAR Wert : CARDINAL;

  PROCEDURE LeerzeichenUeberlesen ...

  PROCEDURE Term() : CARDINAL;
    VAR Wert : CARDINAL;

    PROCEDURE Faktor() : CARDINAL;
      VAR Wert : CARDINAL;

      PROCEDURE CardinalZahl() : CARDINAL;
        VAR Wert : CARDINAL;
        BEGIN
          Wert:=0;
          WHILE (Eingabe[Position]>="0") AND (Eingabe[Position]<="9") DO
            Wert:=10*Wert+ORD(Eingabe[Position])-ORD("0");
            INC(Position)
          END;
          RETURN Wert;
        END CardinalZahl;

      BEGIN (* Faktor *)
        LeerzeichenUeberlesen;
        IF (Eingabe[Position]>="0") AND (Eingabe[Position]<="9")
        THEN RETURN CardinalZahl
        ELSIF Eingabe[Position]="("
        THEN
          INC(Position);
          Wert:=CardinalAusdruck;
          IF Fehler THEN RETURN 0 END;
          LeerzeichenUeberlesen;
```

```
      IF Eingabe[Position]<>")"
      THEN
        Fehlermeldung('")" fehlt!',Position);
        Fehler:=TRUE
      ELSE RETURN Wert
      END
    ELSE
      Fehlermeldung("Illegale Zeichen",Position);
      Fehler:=TRUE
    END (* IF *)
  END Faktor;

BEGIN (* Term *)
  Wert:=Faktor; LeerzeichenUeberlesen;
  WHILE NOT Fehler AND
        ((Eingabe[Position]="*") OR (Eingabe[Position]="/")) DO
    INC(Position);
    IF Eingabe[Position-1]="*"
    THEN Wert:=Wert*Faktor
    ELSE Wert:=Wert/Faktor
    END;
    LeerzeichenUeberlesen
  END;
  RETURN Wert;
END Term;

BEGIN (* CardinalAusdruck *)
  Wert:=Term; LeerzeichenUeberlesen;
  WHILE NOT Fehler AND
        ((Eingabe[Position]="+") OR (Eingabe[Position]="-")) DO
    INC(Position);
    IF Eingabe[Position-1]="+"
    THEN Wert:=Wert+Term
    ELSE Wert:=Wert-Term
    END;
    LeerzeichenUeberlesen
  END;
  RETURN Wert
END CardinalAusdruck;
```

Kapitel 9.2

1. Die Prozedur ist identisch zu «Pos» aus Kapitel 7.4.2, mit dem Unterschied, daß es statt SuchString[i] jetzt SuchString^[i] etc. heißen muß. Anstelle von «StringLaenge» muß selbstverständlich «DynStringLength» verwendet werden.

2.
```
PROCEDURE DynStringEqual(X,Y : DynString): BOOLEAN;
  VAR i,l : CARDINAL;
  BEGIN
    l:=DynStringLength(X);
    IF DynStringLength(Y)<>l
    THEN RETURN FALSE
    ELSE i:=0; WHILE (i<l) AND (X^[i]=Y^[i]) DO INC(i) END;
    RETURN i=l
  END DynStringEqual;
```

3. Wichtig ist der Import von «OpenOutput» aus «InOut». Vor der Ausgabe kann dann die Umleitung erfolgen.

Kapitel 9.3.1.1

1. Höchstwahrscheinlich würde das Programm abstürzen, da unter Umständen in geschützte Speicherbereiche geschrieben wird.
2. Die ganze Änderung besteht darin, daß der Typ CARDINAL durch einen STRING-Typ ersetzt wird, und die entsprechenden Ein- und Ausgabeanweisungen angepaßt werden.

Kapitel 9.3.1.2

```
TYPE Kopf = RECORD
              Inhalt : REAL;
              Rest : Liste
            END;
VAR  Puffer : PufferTyp;
     Messwert : REAL;
...
Puffer.Anfang:=NIL;
LOOP
  IF MesswertDa() THEN
    LiesMessert(Messwert);
    Einfuegen(Puffer,Messwert)
  ELSIF NOT ListeLeer(Puffer) THEN
    Entfernen(Puffer,Messwert);
    BearbeiteMesswert(Messwert)
  END
END (* Loop *)
```

Kapitel 9.3.2

```
VAR Stapel : LIFO;
...
InitLIFO(Stapel);
LOOP
  IF MesswertDa() THEN
    LiesMesswert(Messwert);
    PushToLIFO(Stapel,Messwert)
  ELSIF NOT EmptyLIFO(Stapel) THEN
    PopFromLIFO(Stapel,Messwert);
    BearbeiteMesswert(Messwert)
  END
END (* LOOP *)
```

Kapitel 9.4

```
IF (CAP(S1^[i])=CAP(S2^[i])) AND (S1^[i]<>0C)
THEN INC(i)
ELSE RETURN CAP(S1^[i])<CAP(S2^[i])
END (* IF *)
```

Stichwortverzeichnis

Stichwortverzeichnis