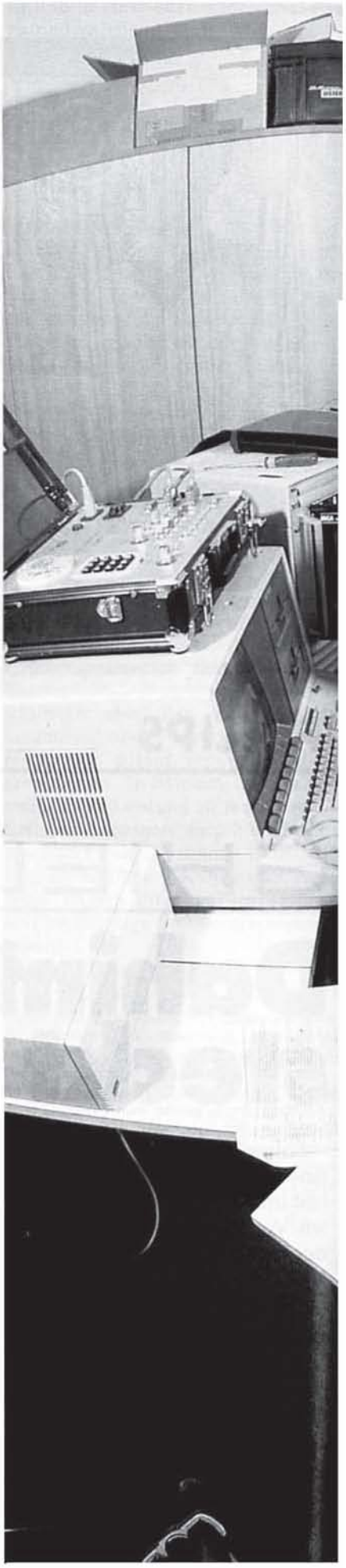
In seinem ersten Teilbeitrag zum Thema Programmiersprachen «Sprachen für die Programmierung — eine Übersicht» in «TR» Nr. 9 vom 26. Februar konzentrierte sich Ludewig auf die frühen Sprachen. Im vorliegenden zweiten Aufsatz soll nun Pascal als Repräsentant der blockorientierten Sprachen das Kernstück bilden, denn an Pascal lassen sich die meisten anderen Konzepte erklären. Was noch fehlt — vor allem die Möglichkeit zur separaten Übersetzung und das Konzept der abstrakten Datentypen —, soll im abschließenden Artikel dieser Reihe in «TR» Nr. 18 vom 30. 4. 1985 bei Modula und Ada nachgetragen werden.

Blockorientierte Programmiersprachen

Von Jochen Ludewig



Nach Vorarbeiten der GAMM (Gesellschaft für Mathematik und Mechanik, BRD) seit 1955 entstand 1960 die Sprache ALGOL 60 (*Algo*rithmic *L*anguage). ALGOL wurde nach drei übergeordneten Zielen entwickelt:

- Mathematische Notation sollte möglichst genau übernommen werden.
- Die Sprache sollte zur Publikation von Algorithmen geeignet sein (FORTRAN wurde für diesen Zweck allgemein als unzureichend empfunden).
- Die Sprache sollte natürlich automatisch in Maschinensprache übersetzbar sein.

Die Urheberschaft von ALGOL geht auf ein Komitee vornehmlich europäischer Wissenschaftler zurück. Diese Herkunft hatte einige bis heute wirksame Konsequenzen:

- Die Sprache wurde für die damalige Zeit außerordentlich sauber konzipiert und definiert. ALGOL war «ein Fortschritt gegenüber den meisten seiner *Nachfolger*» (DAVID GRIES).
- Die Details der Ein- und Ausgabe, die vom algorithmischen Gehalt des Programms her unwichtig sind und sich einer einfachen, formal sauberen Beschreibung entziehen, wurden völlig ausgeklammert.
- Es fehlte die Unterstützung durch die Rechnerhersteller (vor allem IBM).
- ALGOL hat sich in den USA nie durchsetzen können.

ALGOL brachte neben der grundsätzlichen Neuerung einer allgemein als sauber empfundenen Sprache eine ganze Reihe neuer Elemente in die Programmierung ein, darunter vor allem Blockstruktur, Gültigkeitsbereiche, rekursive Prozeduren, strenge Deklarationspflicht, zwei Mechanismen zur Parameterübergabe, formatfreie Syntax und nicht zuletzt ihre formale Definition.
An den europäischen Hochschulen wurde ALGOL 60 vielfach als einzige Programmiersprache gelehrt. Eine große Zahl weiterer Sprachen (die ALGOL-Familie) wurde auf dieser Grundlage entwickelt, darunter EULER, SIMULA, ALGOL W, ALGOL 68 und PASCAL.

Die Entstehung von PASCAL

PASCAL wurde gegen Ende der sechziger Jahre von N. WIRTH und seinen Mitarbeitern an der ETH Zürich geschaffen [9]. Diese Sprache wurde sehr erfolgreich, obwohl hinter ihr weder eine mächtige Organisation wie das Pentagon noch eine große Firma oder eine internationale Expertengruppe stand. Ihre Vorteile bestehen darin, daß sie die Stärken von ALGOL 60 im wesentlichen wahrt, aber zusätzlich einige Konzepte bietet, die man bisher vermißt hatte, nämlich

- Konstanten- und Typdefinitionen
- eine erheblich vergrößerte Vielfalt bei den einfachen und den zusammengesetzten Datentypen (Aufzählungstypen, Records, Zeigervariablen)
- alle Voraussetzungen für die Implementierung relativ einfacher, schneller Compiler, die auch einen effizienten Code erzeugen
- Definition der Ein- und Ausgabe in der Sprache

Das stärkste Merkmal für PASCAL dürfte aber die Einfachheit sein. Wer einmal

Dr. rer. nat. JOCHEN LUDEWIG ist Leiter des Projektes Software Engineering am Brown-Boveri-Forschungszentrum in Baden-Dättwil.

die Prinzipien verstanden hat, braucht beim Programmieren nicht dauernd im Lehrbuch zu blättern, denn die Syntax und die Semantik sind in aller Regel so, wie man dies erwartet. In den letzten Jahren trägt zum Erfolg von PASCAL wesentlich bei, daß die Sprache auch auf sehr kleinen Maschinen (PC) angeboten werden kann.

Deklarationen und Definitionen

Mit FORTRAN war eine Sprache gezeigt worden, in der alle Variablen, Prozeduren und Funktionen automatisch eingeführt werden durch die Verwendung eines neuen Namens; die Festlegung, welchen Typ ein Objekt hat, geschieht dort durch den Kontext. Nehmen wir als Beispiel die folgende Zeile eines FORTRAN-Programms:

```
777 ALPHA = XYZ (17)
```

«777» (vor Spalte 6) ist damit automatisch als Marke identifizierbar. «ALPHA» ist offenbar eine Variable vom Typ REAL (Konvention über Anfangsbuchstaben). «XYZ» muß, wenn es nicht als Feld vereinbart ist, eine Funktion des Typs REAL mit einem INTEGER-Parameter sein.

Im eingeschobenen Beitrag «Exkurs über Softwarefehler» (Seite 70) wird gezeigt, wie diese *implizite Deklaration* Fehler verursachen kann, die sich kaum feststellen lassen. In PASCAL ist sie daher ausgeschlossen.

Ein weiteres, dem Programmierer weniger wichtiges Argument für die vollständige Deklaration ist die Vereinfachung der Übersetzung. Daher wurde bei der Sprache PASCAL an diesem Prinzip auch dort festgehalten, wo die Sicherheit durch die Deklarationen nicht steigt.

Oft werden «Deklaration» und «Definition» verwechselt. Unter *Deklaration* versteht man die Einführung eines Namens, zum Beispiel einer Variablen. Beispiele für Deklaration sind (jeweils in formaler und natürlicher Sprache):

```
LABEL 999;          { 999 ist eine Marke (für Sprunganweisungen)          }
VAR n : integer;    { n ist eine Variable, die als Wert ganze Zahlen hat. }
    p, q : real;    { Die Variablen p und q haben reelle Zahlen als Werte }
```

Eine *Definition* ist dagegen die Angabe einer unveränderlichen Bedeutung für einen Namen. Beispiele:

```
CONST pi = 3.1415;  { pi ist identisch mit der angegebenen Zahl. pi ist
                      also keine Variable !                              }

FUNCTION increment (n: integer): integer;
BEGIN  increment := n + 1  END;
                    { increment ist definiert als eine Funktion, die eine
                      ganze Zahl als Argument erhält und die nächsthöhere
                      ganze Zahl als Ergebnis liefert                    }
```

Softwareentwicklungssystem. Drei Elemente beeinflussen die Arbeit des Programmierers unmittelbar: der Rechnertyp, die Programmiersprache und die Werkzeuge für die Kopplung der beiden ersten Elemente. (Foto: HR. Bramaz)

Viele der Aussagen im Beitrag «Sprachen für die Programmierung» sind für den Laien unverständlich, wenn er dem Problem der Programmfehler nicht ausreichend Beachtung schenkt. Daher seien hier einige Bemerkungen dazu eingeschoben.

Exkurs über Softwarefehler

Wir alle machen ständig Fehler, nicht nur im Bereich der Software. In vielen Situationen fällt uns dies gar nicht auf, da wir daran gewöhnt sind, Fehler sofort zu korrigieren. Zum Beispiel enthält praktisch jede Konversation eine Reihe von Ungenauigkeiten und Fehlern. In der Regel bemerken wir dies sofort und können aus dem Zusammenhang oder durch Fragen die eigentliche Aussage herausfiltern. Ähnlich ist es beim Kochen (Abschmecken) oder beim Autofahren (Lenkkorrektur). In allen diesen Fällen bleibt die Wirkung des Fehlers meist auf den Augenblick beschränkt und kann sofort beseitigt werden.
Anders verhält es sich, wenn die Wirkung erst später sichtbar wird. Beispiele sind Verabredungen oder andere Planungen. Wohl jeder hat in diesem Zusammenhang schon leidvolle Erfahrungen gemacht. Das Problem hat vor allem zwei Ursachen:

- Wir drücken uns ungenau (und damit, strenggenommen, falsch) aus.
- Wir sind nicht in der Lage, komplizierte Vorgänge vollständig zu überblicken.

Daher verabreden wir uns etwa mit einer mißverständlichen Zeitangabe oder an einem Ort, der gerade zu diesem Zeitpunkt unzugänglich ist.
Programme sind äußerst komplizierte Mechanismen; daher haben wir beim Programmieren mehr als bei den meisten anderen Tätigkeiten Gelegenheit, Fehler zu begehen. Da die Ausführung des Programms in der Regel nicht direkt beobachtet wird, erkennen wir Fehler – wenn überhaupt – nur an unerwartetem Verhalten des Programms. Dieses Risiko dürfen wir in vielen Fällen nicht eingehen. Wir müssen daher einen auf den ersten Blick unverständlich großen Aufwand treiben, um Fehler zu *vermeiden*. Da wir dies – wenigstens bis heute – nicht zuverlässig können, ist es außerdem nötig, Fehler zu *erkennen* und so rasch wie möglich zu *beseitigen*.
Untersuchungen verschiedener Autoren (BOEHM, BASILI und PERRICONE) zeigen, daß die durchschnittlichen Kosten eines Fehlers (für Ausfall, Fehlersuche, Beseitigung) exponentiell wachsen mit der Dauer, die der Fehler während der Entwicklung im System war, denn die Folgen breiten sich ja aus.
Diese Feststellungen können in einigen plakativen Sätzen zusammengefaßt werden:

1. *Wir sind zu dumm, um korrekt zu programmieren.*
2. *Das fehlerhafte Programm ist kein Sonderfall, sondern der Normalfall.*
3. *Fehler sind extrem teuer, daher lohnt sich fast jeder Aufwand, um sie zu vermeiden oder zu erkennen.*

Ein FORTRAN-Beispiel

Auf das konkrete Thema «Programmiersprachen» bezogen bedeutet dies, daß wir die Möglichkeit, Fehler zu vermeiden oder zu erkennen, sehr hoch bewerten müssen. Ein bekanntes Beispiel illustriert diese Aussage:
Der erste Venus-Flug der NASA scheiterte, weil in einer Zeile eines großen FORTRAN-Programms statt Komma ein Punkt getippt worden war:

also «DO 9 I = 1.3» (falsch)
statt «DO 9 I = 1,3» (richtig).

Da Blanks (Leerzeichen) bei FORTRAN ohne Bedeutung sind und vor der Übersetzung beseitigt werden, heißt die falsche Version «DO 9 I = 1.3»; dies ist eine Wertzuweisung an die REAL-Variable DO 9 I. Das Programm hat also eine völlig unbeabsichtigte, aber nicht ohne weiteres erkennbare Wirkung.
Ein einfaches und wirksames Verfahren, solche Fehler zu verhindern, besteht darin, vom Programmierer die Deklaration aller Namen zu verlangen, die er verwenden will. Im Falle oben bedeutet dies, daß das Programm entweder eine Deklaration der Variable DO 9 I enthält oder fehlerhaft ist (was zutrifft). Im letzteren Fall bekommt der Programmierer eine Fehlermeldung des Compilers und findet den Fehler mühelos. Der Zwang zur Deklaration (das heißt Verzicht auf die zunächst bequeme automatische Einführung) schafft also einen Schutz gegen triviale, aber schwer erkennbare Fehler.
Daher sollte jede Programmiersprache unbedingt die vollständige Deklaration erzwingen. FORTRAN-Programmierer sollten beachten, daß sie mit FORTRAN-77 diesen Effekt annähernd erzielen können durch die Angabe IMPLICIT NONE, das heißt durch das Ausschalten der voreingestellten Typzuordnung.

Ein PL/I-Beispiel

Bei *PL/I* handelt es sich um eine Mischung der Konzepte aus FORTRAN, COBOL und ALGOL 60, die von IBM seit 1964 entwickelt und lanciert wurde. Diese wegen ihrer Mächtigkeit bei vielen Programmierern beliebte Sprache ist bekannt für ihre *Fallen*, also Hindernisse, denen wir zum Opfer fallen, ohne sie vorher zu erkennen. Der nachfolgende Programmausschnitt zeigt ein Beispiel.
Seien A und B quadratische Matrizen der Ordnung 2 mit folgenden Werten:

$$A = \begin{pmatrix} 2 & 4 \\ 3 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 5 \\ 7 & 8 \end{pmatrix}$$

Gesucht ist das Ergebnis, das der folgende Befehl liefert:

A = (A + B) ** 2 – A (1,1)

Der *PL/I*-Laie wird sich zumindest über die Subtraktion wundern, vielleicht weiß er, daß sie für alle Elemente der Matrix vorgenommen wird. Er wird dann wahrscheinlich wie folgt rechnen: Zunächst werden A und B als Matrizen addiert und anschließend quadriert; nachher wird von allen Elementen des Resultats A (1,1), also 2, abgezogen.
Diese Rechnung weicht zweifach von der «offiziellen» Bedeutung des Programms ab: Erstens erfolgt das Quadrieren nicht für die Matrix insgesamt, sondern elementweise. Zweitens hat die Subtraktion einen Nebeneffekt; nachdem sie beim ersten Element der Matrix ausgeführt ist, hat A (1,1) einen neuen Wert (nämlich 7), der für die drei weiteren Subtraktionen verwendet wird.
Problematisch an einer solchen Sprachdefinition ist, daß es in einem größeren Projekt mit Sicherheit Leute gibt, die alle Finessen einer Sprache kennen und ausnutzen, und andere, die die Sprache nur oberflächlich

kennen und nicht in der Lage sind, die Tricks der anderen zu durchschauen. Da die Konstruktion oben nicht einfach unverständlich ist, sondern eine falsche Bedeutung suggeriert, wird ein Mitarbeiter der zweiten Kategorie das Programm arglos verändern in der Annahme, es verstanden zu haben.

Der Wert von Tests

Vielfach hört man die Meinung, Fehler könnten ja einfach durch Testen der Programme entdeckt und dann beseitigt werden. Diese Vorstellung ist falsch. Wenn ich in die USA reise, kann ich hinterher nicht behaupten, daß keiner meiner Bekannten je dort war, nur weil ich keinen getroffen habe. Nur die positive Aussage ist möglich: Treffe ich zufällig einen Freund, so weiß ich, daß er dort war. Entsprechend kann Testen nur die *Anwesenheit* eines Fehlers zeigen, nicht die *Abwesenheit* (Dijkstra). Die Zahl der möglichen Ausführungen eines Programms ist mehr als astronomisch. In diesem Sinne ist der Vergleich oben (eine bestimmte Person zwischen 200 Millionen Menschen treffen) stark verharmlosend!

Natürlich können wir unsere Chancen verbessern, indem wir uns an der Freiheitsstatue oder an der Golden Gate Bridge postieren, aber eine Garantie ist dies noch lange nicht. Im gleichen Sinne wählen wir beim Testen Fälle aus, die besonders kritisch erscheinen, aber vielleicht versagt das Programm gerade mit einer bestimmten Kombination von sehr gewöhnlich aussehenden Daten. Testen ist also eine Krücke mit äußerst beschränkter Tragfähigkeit.

«Aber die Hardwareleute, die können doch auch vollständig testen!» Auch diese Annahme ist falsch, wie eine simple Rechnung zeigt. Nehmen wir an, ein Addierwerk für 32 Bits soll getestet werden. Offenbar gibt es für jeden der beiden Summanden 2^{32} verschiedene Werte, insgesamt also 2^{64} mögliche Additionen. Bei einem sehr schnellen Testsystem (100 Millionen Tests pro Sekunde, also einer in 10 ns) dauert der vollständige Test etwa 6000 Jahre (!). Daß die Hardware in der Regel sinnvoll arbeitet, liegt also nicht am Testen, sondern an der (zum Teil durch den Werkstoff erzwungenen) Disziplin der Elektroniker. Unser Werkstoff, die Programmiersprache, sollte ähnliche Eigenschaften haben.

In der Regel impliziert die Definition die Deklaration. Ausnahmen gelten dort, wo ein Name schon vor der Definition verwendet wird. In diesem Fall müssen auch die Namen von Objekten, die später definiert werden, vorher deklariert sein. Dieser Fall tritt beispielsweise auf, wenn sich zwei Prozeduren gegenseitig aufrufen.

Rekursion

Nachfolgend soll auf den Begriff der Rekursion als fundamentales Prinzip der Informatik näher eingegangen werden.

Die einfachste Definition der Rekursion sieht so aus:

Rekursion: siehe Rekursion

Rekursion ist das Grundprinzip des Lebens schlechthin. Jede Vermehrung von Pflanzen oder Tieren findet in der Natur rekursiv statt, indem die Nachkommen selbst Nachkommen haben. Man beachte den Unterschied zwischen Rekursion und *Iteration:* Bei der Iteration erzeugt ein einziger Mechanismus viele gleiche Objekte (zum Beispiel: eine Maschine erzeugt immer wieder den gleichen Kamm).

In der Informatik ist die Rekursion schon durch RUTISHAUSER 1952 theoretisch eingeführt worden, aber erst mit der Sprache ALGOL 60 wurde sie praktisch angewendet. Anders als in FORTRAN ist es in ALGOL 60 (und danach in allen Sprachen der ALGOL-Familie) erlaubt, daß sich eine Prozedur selbst aufruft. Dies kann direkt oder über den Umweg weiterer zwischengeschalteter Prozeduraufrufe geschehen.

Wir wollen nun eines der klassischen Beispiele für eine rekursive Prozedur betrachten, die Lösung der Aufgabe, die den «Türmen von Hanoi» zugrunde liegt. Gegeben ist ein Turm von Scheiben, deren Größe von oben nach unten zunimmt.

Die Aufgabe besteht nun darin, diesen Turm abzutragen und an anderer Stelle aufzubauen, wobei folgende Einschränkungen gelten sollen:

- Es darf jeweils nur eine einzige Scheibe bewegt werden.
- Es dürfen nie mehr als drei Türme bestehen.
- Alle Türme müssen zu jeder Zeit, wie oben gezeichnet, geordnet sein.

Das folgende Bild zeigt eine Momentaufnahme aus dem Vorgang:

Offenbar muß nun zunächst die Spitze (der Turm rechts) auf den mittleren Turm gebracht werden, dann kommt die große Scheibe nach rechts, und der Turm in der Mitte wird mit sieben weiteren Bewegungen auf die große Scheibe gebracht.

Mit rekursiven Prozeduren können die notwendigen Bewegungen nun sehr elegant beschrieben werden. Hier ist das vollständige PASCAL-Programm dafür:

```
PROGRAM hanoi (output),

CONST   gesamthoehe = 4,

TYPE    resthoehe = 0 .gesamthoehe,
        plaetze = (ALPHA, OMEGA, DELTA),

PROCEDURE bewegetaem (hoehe, resthoehe, plats1, plats2, plats0, plaetze),

BEGIN { bewegetaem }
  IF hoehe > 0
  THEN                                                              BEGIN
     bewegetaem (hoehe - 1, plats1, plats0, plats2),
     writeln ('Scheibe ', hoehe:1, ' von ', plats1, ' nach ', plats2);
     bewegetaem (hoehe - 1, plats0, plats2, plats1),                 END,
END { bewegetaem };

BEGIN { hanoi }
   bewegetaem (gesamthoehe, ALPHA, OMEGA, DELTA);
END { hanoi }.
```

Dieses Programm soll nun erläutert werden.

Die Gesamthöhe des Turms ist hier als Konstante definiert. Statt dessen könnte natürlich auch eine Variable deklariert werden, die vom Benutzer des Programms besetzt wird (dann muß im Programmkopf neben «output» auch «input» angegeben werden). Man beachte, daß im Programm oben keine einzige Variable vorkommt.

Die «resthoehe» und die «plaetze» werden als Typen definiert. Die «resthoehe» soll ein Wert zwischen 0 und der Gesamthöhe sein; dadurch, daß dafür nicht einfach Integer-Werte verwendet werden, können Fehler zur Übersetzungs- oder zur Laufzeit des Programms eventuell entdeckt werden. Die Plätze werden mit drei Phantasienamen bezeichnet. Es handelt sich also um einen Aufzählungstyp.

Das Hauptprogramm besteht nur aus einer (der vorletzten) Zeile: Es verlangt, daß ein Turm der angegebenen Höhe von Platz ALPHA nach Platz OMEGA gebracht wird. Dabei kann der Platz DELTA zu Hilfe genommen werden.

Die dazu aufgerufene Prozedur ist nicht viel komplizierter: Verlangt man von ihr den Transport eines Turms der Höhe Null, also eines nicht vorhandenen Turms, so tut sie natürlich gar nichts. Andernfalls transportiert sie den oberen Teil auf den Hilfsplatz, die unterste Scheibe auf den Zielplatz und schließlich den oberen Teil vom Hilfsplatz auf den Zielplatz. Der Transport der einzelnen Scheibe hat im Ablauf weiter keine Wirkung, er wird nur durch die write-Anweisung in der Ausgabe vermerkt.

Die Abbruchbedingung (hier: Abfrage ob «hoehe» > 0) ist ein notwendiges

Merkmal jeder rekursiven Prozedur, ohne sie würde die Rekursion immer weiter fortschreiten und nie zum Ende kommen (wie es bei der biologischen Rekursion der Fall ist).
Eine Ausführung des Programms gibt folgendes Ergebnis:

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

Wer mit rekursiven Prozeduren nicht vertraut ist, kann sich den Ablauf in verschiedenen Ebenen aufzeichnen. Man muß sich klarmachen, daß mit einem Prozeduraufruf die Ausführung der *aufrufenden* Prozedur zugunsten der *aufgerufenen* unterbrochen wird. Dies gilt auch, wenn – wie im Fall der rekursiven Prozedur – beide *statisch* identisch sind. *Dynamisch* existieren sie nebeneinander, jede *Inkarnation* mit eigenen Parametern und allenfalls Variablen. In unserem Beispiel existieren also, wenn die erste Zeile («Scheibe 1 von . . .») ausgegeben wird, folgende Inkarnationen der Prozedur «bewegeturm»:

```
bewegeturm  ( 4, ALPHA, OMEGA, DELTA )
bewegeturm  ( 3, ALPHA, DELTA, OMEGA )
bewegeturm  ( 2, ALPHA, OMEGA, DELTA )
bewegeturm  ( 1, ALPHA, DELTA, OMEGA )
```

Während die unterste die Ausgabe erzeugt, warten alle anderen auf das Ende der jeweils tieferen.
Natürlich funktioniert das Programm genauso, wenn der Wert von «gesamthoehe» größer gewählt wird. Die Ausgabe wird dann allerdings leicht recht lang: Für 20 Scheiben ergeben sich schon über eine Million Zeilen, ausgedruckt etwa ein Papierstapel von 2 m Höhe, denn die Zahl der Bewegungen steigt exponentiell nach der Formel (2^n-1).
Die rekursive Beschreibung eignet sich überall dort, wo man geschachtelte Strukturen bearbeitet oder die Strukturen zumindest so auffassen kann. Beispiele sind etwa die Übersetzung von rekursiv definierten Sprachen (ALGOL-Familie), die Suche in baumartig organisierten Daten, die Programmierung eines automatischen Spielpartners (die Rekursivität liegt hier im Bestreben, die Planung des Gegners in die eigene Planung einzukalkulieren).
Es ist immer wieder verblüffend zu sehen, wie einfach scheinbar höchst komplizierte Probleme werden, wenn sie rekursiv beschrieben sind. (Zweifler sind eingeladen, die Türme von Hanoi einmal in FORTRAN zu programmieren. Es ist mühsam, selbst dann, wenn man ein rekursives Programm als Vorlage benutzt.)

Das Prinzip der Blockstruktur, Kellertechnik

FORTRAN-Programme liegen logisch auf einer einheitlichen Stufe, sie bilden sozusagen formal eine Ebene. Bei der Programmierung ist es aber oft sinnvoll, einen Teil des Programms nur ganz *lokal* zu betrachten und nach außen, gegen andere Programmteile, nicht im Detail sichtbar werden zu lassen. Ursprünglich spielte dabei auch das Argument eine Rolle, daß man auf diese Weise Speicherplatz sparen kann.
Ein Programm mit Blockstruktur kann man daher als System geschachtelter Programmteile auffassen. Nehmen wir etwa an, daß im Beispiel der Türme von Hanoi noch zwei Erweiterungen vorgenommen seien, die beide einer verbesserten Präsentation der Ergebnisse dienen: Zu Beginn sei eine Erläuterung auszudrucken, und die Bewegungen der Scheiben sollen in einer etwas anschaulicheren Darstellung (zum Beispiel durch Pfeile) gezeigt werden. Dazu wird jeweils eine weitere Prozedur vorgesehen, «inittext» und «zeigebewegung». «zeigebewegung» wird nur innerhalb von «bewegeturm» benötigt und kann daher dort eingeschachtelt sein. Die nackte Struktur mit jeweils einer Deklaration in jeder Prozedur sieht dann insgesamt so aus:

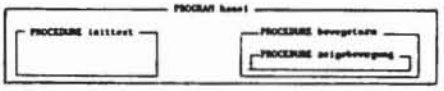
```
PROGRAM hanoi (output).

  PROCEDURE inittext;
    VAR zeile  PACKED ARRAY [1 .. 80] OF char.
  BEGIN    END.

  PROCEDURE bewegeturm
    VAR i : integer.
    PROCEDURE zeigebewegung.
      VAR j  integer.
    BEGIN { zeigebewegung }    END { zeigebewegung }.
  BEGIN { bewegeturm }    END { bewegeturm }.

BEGIN { hanoi }    END { hanoi }
```

Die Schachtelungsstruktur, die durch die Einrückung angedeutet ist, sieht also so aus:



Namen sind grundsätzlich *nur* gültig innerhalb der Schachtelung, in der sie definiert oder deklariert werden; man nennt diesen Programmabschnitt ihren *Gültigkeitsbereich*. «i» und «zeigebewegung» sind also *lokal* zu «bewegeturm» und können nur dort, nicht in «inittext» oder im Hauptprogramm «hanoi», verwendet werden.
Der von der Variablen «zeile» belegte Speicherplatz kann, wenn die Ausführung von «inittext» beendet ist, von «i» in «bewegeturm» erneut belegt werden, denn der Name «zeile» ist ohnehin nicht mehr verwendbar. Daraus ergibt sich das Prinzip des *Kellerspeichers* (*Push-down-stack* oder einfach Stack). Jede Prozedur belegt den von ihr zusätzlich benötigten Speicherplatz nur so lange, wie sie selbst oder eine untergeordnete Prozedur ausgeführt wird. Danach sind alle *lokalen Variablen* verloren. Es ist dabei auch gestattet, in einer eingeschachtelten Prozedur dieselben Namen erneut zu belegen. Würde also etwa in «zeigebewegung» ebenfalls ein «i» verwendet, so wäre das nicht weiter schlimm, nur ist das «i» aus «bewegeturm» nicht zugänglich (aber noch vorhanden), solange «zeigebewegung» läuft.
Dieses Kellerprinzip, das in Zusammenhang mit ALGOL 60 entstanden ist, hat nun neben der Speicherplatzeinsparung noch Vorteile, die heute wichtiger sind. Insbesondere lassen sich so auch rekursive Prozeduren (wie die Prozedur «bewegeturm») recht einfach realisieren. Bei jedem rekursiven Aufruf wird eine neue Schicht im Keller angelegt, in der «i» und alle anderen lokalen Variablen erneut einen Speicherplatz erhalten. Ist also «bewegeturm» vierfach rekursiv aufgerufen, so existieren vier Inkarnationen der Prozedur als Schichten im Keller. Der Keller wird auch für die Parameterübergabe und die Speicherung von Zwischenergebnissen benutzt. Seit den sechziger Jahren werden auch Maschinen gebaut, die spezielle Hardware für die Organisation des Kellers besitzen.

Schrittweise Verfeinerung

Schon DESCARTES hat beschrieben, wie man ein komplexes Problem lösen könne: Man zerlegt es in Teile und zerlegt diese weiter, bis die Teile klein genug sind, um einfach lösbar zu sein. Dieses rekursive Prinzip läßt sich auf die Programmierung übertragen [10]. Man spricht hier von schrittweiser Verfeinerung («stepwise refinement»). Am Beispiel eines Programms, das Texte für den Druck aufbereitet, soll das Verfahren erläutert werden:
Die – zunächst noch unklare – Problemlösung wird als Programm konzipiert und niedergeschrieben, also etwa

```
PROGRAM aufbereitung;
BEGIN

END
```

Nun werden die Hauptaufgaben festgelegt: Zunächst muß der Text aus irgendeiner Quelle geholt werden, dann wird er aufbereitet und schließlich ausgegeben. Der noch leere Rumpf des Programms wird also gefüllt mit den drei Prozeduraufrufen

```
holetext  (rohtext),
bereitsauf (rohtext, fertigtext),
gibtextaus (fertigtext),
```

«rohtext» und «fertigtext» sind hier Parameter, denn diese Information wird jeweils weitergegeben, kann also nicht *lokal* innerhalb der drei Prozeduren behandelt werden. Bevor weiter verfeinert wird, muß also die Struktur dieser Variablen durch eine Deklaration festgelegt werden. Wir gehen hier davon aus, daß es sich syntaktisch einfach um Textdateien handelt. Im Realfall muß natürlich jetzt auch die logische Struktur festgelegt werden, wir lassen das hier weg. Nach demselben Schema wird die Verfeinerung nun fortgesetzt:

```
PROCEDURE bereitsauf (VAR rohtext, fertigtext: text);
VAR  rohabsatz, absatz   text,
BEGIN
   WHILE NOT eof (rohtext) DO                    BEGIN
      absatzholen (rohtext, rohabsatz);
      absatzaufbereiten (rohabsatz, absatz);
      absatzkopieren (absatz, fertigtext)        END
END;
```

Schließlich könnte die Prozedur «absatzholen» so aussehen:

```
PROCEDURE absatzholen (VAR textein, textkopie  text);
BEGIN
   rewrite (textkopie);
   WHILE NOT (eof (textein) OR eoln (textein)) DO
      holezeile (textein, textkopie);
END;
```

Man sieht, daß das Problem auf jeder Stufe teilweise durch die Operationen der virtuellen Maschine (zum Beispiel rewrite-Statement), teilweise durch untergeordnete Prozeduren, die noch zu realisieren bleiben, gelöst wird.
Offenbar dient hier die Unterprogrammtechnik nicht mehr primär zur Speicherplatzeinsparung, also zur Unterstützung der Maschine, sondern zur Verminderung der Komplexität, also zur Unterstützung des Programmierers.
Im Zusammenhang mit dem Begriff der virtuellen Maschinen ist bekannt, daß sich mit der Realisierung einer virtuellen Maschine durch eine andere Schichten bilden. Ähnliches geschieht hier. Auf der obersten Ebene liegt nur die Bezeichnung des zu lösenden Problems. In den folgenden Schichten wird diese Bezeichnung immer stärker aufgefüllt mit den Details der Lösung durch die Operationen der virtuellen Maschine. Daher spricht man auch von einem *Top-Down-Verfahren.* Den Gegensatz *(Bottom-Up)* bildet ein Vorgehen, das mit den Operationen der virtuellen Maschine beginnt und diese zu immer komplexeren Operationen kombiniert.
Beide Verfahren sind idealisiert, in der Wirklichkeit können wir nur mit einer Kombination leben. Dabei ist es in der Regel besser, den Top-Down-Ansatz zu betonen.

Parameterübergabe

Beim Anruf einer Prozedur (einschließlich von Funktionen) werden dieser in der Regel *Parameter* übergeben, und zwar solche, die die Werte liefern, mit denen die Prozedur rechnen soll, und auch solche, die die Ergebnisse zurücktransportieren sollen. Das folgende Primitivbeispiel (besser lösbar durch eine Funktion) zeigt beide Varianten am Beispiel einer FORTRAN-Subroutine, die die Wurzel aus dem Absolutwert einer Zahl zieht.

```
      SUBROUTINE ABSSQRT (X, Y)
      IF (X) 10, 20, 20
   10 X = -X
   20 Y = SQRT (X)
      RETURN
      END
```

X ist hier der Eingabeparameter, Y der Ausgabeparameter. Wird diese Prozedur im Programm verwendet, so sind Komplikationen wahrscheinlich, denn FORTRAN-Systeme realisieren die Parameterübergabe auf eine einheitliche, recht einfache Art («call by reference»); praktisch arbeitet die Subroutine mit den Variablen genau wie das aufrufende Programm. Daher ist X, falls es zuvor negativ war, nach der Ausführung von ABSSQRT verändert. Solche Effekte führen zu schwer erkennbaren Fehlern. (Besonders unangenehm ist diese Nebenwirkung, wenn im Aufruf eine Konstante als Parameter steht, zum Beispiel CALL ABSSQRT (–4.0,A). Erscheint dieselbe Konstante an weiteren Stellen des Programms, so wird dort mit +4.0 statt –4.0 gerechnet.)
In PASCAL gibt es daher zwei verschiedene Übergabemechanismen, wobei der unsichere (wie oben) nur verwendet wird, wenn der Programmierer dies ausdrücklich verlangt. Der andere ist ein «call by value»: Das aufgerufene Programm erhält nicht X, sondern eine Kopie davon.

```
PROCEDURE absqrt (X  real; VAR Y: real);
BEGIN
   IF (X < 0)
   THEN X  = - X,
   Y := sqrt (X);
END;
```

Diese Prozedur verändert X im aufrufenden Programm nicht. Allerdings wäre praktisch eine bessere Lösung die mit

```
IF (X < 0)
THEN Y := SQRT (-X)
ELSE Y := SQRT ( X)
```

wobei das Problem nicht mehr auftritt.
In ADA gibt der Programmierer explizit an, ob es sich um einen IN-, OUT- oder INOUT-Parameter handelt. Der sicherste Mechanismus (IN = call by value) wird gewählt, wenn eine Angabe fehlt.

Konstanten- und Typvereinbarungen

Konstanten

Enthält ein Programm eine Zahl, so kann der Leser nicht ohne weiteres feststellen, welche Bedeutung diese Zahl hat. Dies erschwert das Verständnis und bildet bei Programmänderungen ein zusätzliches Risiko. Nehmen wir etwa an, ein FORTRAN-Programm arbeite mit den Arbeitszeiten von fünf Mitarbeitern, indem es sie in einer Tabelle ablegt, für jeden Arbeitstag eine Spalte, also in Form einer 5 × 5-Matrix. Die Initialisierung dieser Matrix sieht so aus:

```
      DIMENSION ZEIT (5, 5)

      DO 10  I = 1, 5
      DO 10  J = 1, 5
   10 ZEIT (I, J) = 0.0
```

Wird nun ein weiterer Mitarbeiter eingestellt, so muß eine Zahl geändert werden; wenn der Programmierer das Programm nicht gut kennt, sieht es sehr wahrscheinlich hinterher so aus:

```
      DIMENSION ZEIT (6, 6)
      ..
      DO 10  I = 1, 6
      DO 10  J = 1, 6
   10 ZEIT (I, J) = 0.0
```

Die Zahl «5» läßt nicht erkennen, ob sie die Zahl der Mitarbeiter oder die der Arbeitstage bezeichnet. Daher ist die Verwendung von Zahlen im Programm stets eine Quelle von Mißverständnissen. In PASCAL gibt es die Möglichkeit, Konstanten zu definieren und diese im Programm zu verwenden. Das Beispiel oben sieht damit wie folgt aus:
Im Programmkopf:

```
CONST Arbeitstage = 5,
      Mitarbeiter = 5,

VAR   Zeit   ARRAY [1..Arbeitstage, 1..Mitarbeiter] OF real;
      i, j   integer;
```

Im Programmrumpf:

```
FOR i  = 1 TO Arbeitstage DO
   FOR j := 1 TO Mitarbeiter DO
      Zeit [i, j] := 0.0;
```

Die Vorteile sind offensichtlich: Das Programm braucht nur an einer einzigen, leicht erkennbaren Stelle geändert zu werden und wird nicht durch Mißverständnisse gefährdet. Da die symbolischen Konstanten schon vom Compiler durch ihre Werte ersetzt werden, erhöht sich die Laufzeit des Programms nicht.

Primitive und komplexe Typen

PASCAL kennt zunächst die Typen *integer, real, boolean* (mit den Werten true und false) und *char* (mit allen Zeichen des Zeichensatzes als Werten, zum Beispiel ‹X› oder ‹;›).
In den älteren Programmiersprachen gibt es meist nur eine kleinere Menge vordefinierter Typen, mit denen der Programmierer auskommen muß. Aus diesen *primitiven Typen* können auch Felder und (in COBOL) komplexe Strukturen aufgebaut werden. Es ist nun praktisch, wenn man nicht nur einzelne

Variablen einer solchen komplexen Struktur vereinbaren kann, sondern generell einen neuen Typ, der dann – wie die Konstanten oben – den Verwendungszweck erkennen läßt und für mehrere Variablen und Prozedurparameter verwendet werden kann. In PASCAL besteht, zusammen mit dem erweiterten Bestand primitiver Typen und Konstrukten zum Aufbau komplexer Typen, diese Möglichkeit. Dadurch wird das Programm wesentlich übersichtlicher und unempfindlicher gegen Änderungen.

Aufzählungstypen und Mengen

In der Realität, die durch ein Programm beschrieben werden soll, haben wir es oft mit Mengen zu tun, deren Elemente durch unterschiedliche Bezeichnungen unterschieden und eventuell geordnet sind (das heißt, es gibt eine vorgegebene Reihenfolge), aber keine Rechengrößen darstellen. Beispiele sind die Farben Rot, Gelb, Grün einer Lichtsignalanlage, die Teilnehmer eines Schachspiels (Weiß, Schwarz) oder einer Bridge-Partie (Nord, Ost, Süd, West). Für solche Objekte gibt es in PASCAL die sogenannten *Aufzählungstypen:*

TYPE spieler = (Nord, Ost, Süd, West).

Bei den Türmen von Hanoi waren die drei Plätze als ein solcher Aufzählungstyp deklariert worden. Bietet die Programmiersprache diese Möglichkeit nicht, so kann man natürlich auf Zahlen ausweichen, zum Beispiel mit der Konvention ALPHA = –1, DELTA = 0, OMEGA = 1. Dies führt aber leicht zu Fehlern, ein anderer beteiligter Programmierer könnte beispielsweise irrtümlich gerade die umgekehrte Codierung benutzen oder die Zahlen 1, 2, 3.
Vielfach kommen auch für Zahlen oder andere Symbole nur ganz bestimmte Werte in Frage, so etwa für die Variable «hoehe» in «Hanoi». Durch die Möglichkeit, den Typ einzuschränken, erhalten Übersetzer und Laufzeitsystem die Möglichkeit, gewisse Fehler automatisch zu erkennen.
Aus Objekten eines Aufzählungstyps können *Mengen* gebildet werden, auf die die üblichen Mengenoperationen (Vereinigung, Schnitt, Differenz) definiert sind. Auch dies verbessert die Lesbarkeit eines Programms erheblich. Ein Zeichen soll zum Beispiel darauf getestet werden, ob es sich um eine Ziffer handelt:
IF zeichen IN ['0' .. '9'] THEN ...
Die eckige Klammer definiert eine (in diesem Beispiel konstante) Menge. Entsprechend können auch Mengenvariablen vereinbart und verwendet werden.

Komplexe Datentypen

In PASCAL gibt es vier Möglichkeiten, aus Daten komplexere Gebilde aufzubauen:

- Felder (Arrays) ähnlich wie in FORTRAN, aber mit beliebigen Aufzählungstypen zur Indizierung, also etwa auch «ARRAY [spieler] OF integer»
- Verbunde (Records) ähnlich wie in COBOL, aber mit wesentlich größeren Kombinationsmöglichkeiten und mit Variantenfeldern
- Dateien (Files), die den Feldern ähnlich sind, da aus einem einzigen Grundtyp eine Reihe von Objekten entsteht. Files werden aber anders verwaltet, so daß einerseits die Indizierung entfällt, anderseits beliebig viele Elemente möglich sind. Dateien dienen vor allem zur Ein- und Ausgabe von Daten (permanente Files), aber auch zur vorübergehenden Speicherung während des Programmlaufs (temporäre Files)
- logische Geflechte, die durch die Verwendung von Zeigern entstehen

Felder und Verbunde können auch in jeder beliebigen Form geschachtelt werden. Man erkennt hier einen wesentlichen Vorteil des rekursiven Ansatzes; Einschränkungen, wie es sie in FORTRAN gibt (beispielsweise auf maximal drei Indizes), würden die Implementierung eines PASCAL-Compilers nicht erleichtern, sondern erschweren.
Auf die Felder, Verbunde und Dateien soll hier nicht weiter eingegangen werden, es folgen nur ein paar Beispiele.

```
TYPE    Namtyp = ARRAY [1..30] OF char;          { Feld aus 30 Zeichen }

        Datum  = RECORD  Tag   : 1..31;     { Verbund aus drei Teilen }
                         Monat : 1..12;
                         Jahr  : 0..99,
                 END;

        Personendaten = RECORD  Name          Namtyp;   { Verbund aus }
                                Geburtsort :  Namtyp;   { Feldern und }
                                Geburtstag :  Datum;    { Verbund.    }
                        END;
VAR     Tagesdatum : Datum;
        Mitarbeiter : FILE OF Personendaten;  { Datei aus Verbund-Obj.}
        Neueinstellung : Personendaten,       { einzelner Verbund     }
```

Zeigertypen

Bei komplizierten logischen Strukturen tritt oft das Problem auf, daß Objekte untereinander verknüpft sind. Nehmen wir etwa an, daß wir die verwandtschaftlichen Beziehungen zwischen vielen Menschen speichern wollen. In einer Kartei tun wir dies mit Verweisen («W. F. BACH, Sohn von → J. S. BACH, Bruder von → C. PH. E. BACH»). In einer vom Rechner verwalteten Datenstruktur wäre es nun unzweckmäßig, diese Verweise genauso explizit zu verwenden, denn wir müßten im Falle oben erst wieder den betreffenden Namen suchen. Statt dessen verweisen wir lieber direkt auf die betreffende «Karteikarte». In PASCAL gibt es dafür *Zeigervariablen*. Mit Verbunden (Records), in denen mindestens eine Komponente ein Zeiger ist, lassen sich beliebig umfangreiche logische Netze aufbauen und bearbeiten.
Diese Netze verändern sich während des Programmlaufs dynamisch, ihre Größe, das heißt der belegte Speicherplatz, läßt sich also nicht a priori festlegen. Damit sprengen sie das Prinzip des Kellerspeichers; für sie wird (logisch) ein zweiter Speicher, die sogenannte *Halde (Heap)*, vorgesehen. Die dort abgelegten Daten können aus Konsistenzgründen *nur* über Zeiger erreicht werden; für ihre Schöpfung und Beseitigung gibt es zwei spezielle Operationen «new» und «dispose». Die Verwaltung der Halde, auf die hier nicht näher eingegangen werden kann, ist komplizierter als die eines Kellerspeichers.
Als Beispiel für Zeigervariablen ist hier ein einfaches Programm angegeben, das eine Liste von Namen (jeweils ein Buchstabe) einliest und diese als eine Zeigerkette speichert, bis ein Leerzeichen kommt. Interaktiv werden dann aus der Kette die Elemente einzeln wieder entfernt, bis sie leer ist. Man beachte, daß mit dem Entfernen in keinem Fall ein Umspeichern verbunden ist, es wird nur der Anfangszeiger oder der des Vorgängers umgehängt, zum Beispiel so:
vorher:
erster → A → B → C → D → NIL
nachher:
erster → A → B ——→ D → NIL
Zum Verständnis des Programms vorweg einige kleine Beispiele, die den Ge-

brauch des senkrechten Pfeils in PASCAL zeigen:

```
x,y   ↑integer,      { Zwei Zeiger-Variablen x und y im Keller ein, die
                       auf eine integer-Variable in der Halde zeigen können}
new (x);             { lassen eine integer-Var  entstehen, ( Ego  ist
                       disposo )                                          }
x↑  = 17;            { weist der integer-Variablen den Wert 17 zu         }
y   = x;             { lassen y auf dieselbe integer-Variable zeigen      }
```

Und hier ist das Programm «Kette»:

```
PROGRAM kette (input, output),

TYPE   link    = ↑ element,              { Vorwärts-Referenz hier zulässig }
       element = RECORD name      : char;
                        nachfolger  link,
                 END;
VAR    erster  link;                      { Zeiger zum ersten in der Kette }
{-----------------------------------------------------------------------}

PROCEDURE bearbeite,                                   { zur Eingabe einer Kette }

VAR    letzter  link,                                  { Zeiger auf Ende der Kette }

  PROCEDURE belesenen (VAR eintrag  link),             { Eingabe eines Namens }
  VAR  zeichen : char;     { Die Namen bestehen einfach aus einem Buchstaben }
  BEGIN
    write ('Gib Namen   '), readln (zeichen),
    IF zeichen = ' '                                   { Ende der Eingabe durch Blank }
    THEN
      eintrag := NIL
    ELSE                                                          BEGIN
      new (eintrag),                          { neuen Element fuer Liste schaffen }
      eintrag↑  name := zeichen,              { Namen einsetzen }   END
  END { belesenen },

BEGIN { bearbeite }
  belesenen (erster);                                              { Initialisierung }
  letzter := erster;
  WHILE letzter <> NIL DO   { Ende noch nicht erreicht }          BEGIN
    belesenen (letzter↑  nachfolger),     { Parameter ist nicht initialisiert }
    letzter  = letzter↑  nachfolger       { Zeiger weitere  }  END
END { bearbeite },
{-----------------------------------------------------------------------}

PROCEDURE printkette,                                          { zur Ausgabe der Kette }

VAR    aktueller  : link;
BEGIN
  writeln; write ('Stand: ');
  IF erster = NIL
  THEN
    writeln ('Die Kette ist leer')
  ELSE                                                             BEGIN
    aktueller  = erster,
    REPEAT
      write (' - ', aktueller↑ name);
      aktueller := aktueller↑  nachfolger,
    UNTIL aktueller = NIL;
    writeln (' -')     { Zeilen-Abschluss der Ausgabe }          END;
END { printkette };
{-----------------------------------------------------------------------}

PROCEDURE loeschen,     { zum Loeschen eines eingegebenen Elements der Liste }

VAR    loeschname : char;
       vorgaenger, naechster   link,
BEGIN                                          { Kette ist definitiv nicht ganz leer }
  write ('                                          zu loeschender Name: ');
  readln (loeschname);

  naechster := erster,          { naechster zeigt zunaechst auf ersten Element }
  WHILE (naechster↑  name        <> loeschname)
    AND (naechster↑  nachfolger <> NIL)          DO  { suchen }      BEGIN
      vorgaenger := naechster,                  { Vorgaenger merken }
      naechster  := naechster↑. nachfolger                            END,

  IF (naechster↑  name = loeschname)    { d.h. falls loeschname in der Liste }

  THEN { Nachfolger an den Vorgaenger haengen }                     BEGIN
    IF naechster = erster                          { d.h. falls erstes Element wegfaellt }
    THEN erster                  := naechster↑  nachfolger
    ELSE vorgaenger↑  nachfolger  = naechster↑  nachfolger;
    dispose (naechster), { loeschname freigeben }                     END

  ELSE
    writeln ('*** ', loeschname, ' ist nicht in der Kette enthalten ***'),
END { loeschen },
{-----------------------------------------------------------------------}

BEGIN { kette }
  bearbeite;
  printkette;
  WHILE erster <> NIL DO     { Kette ist nicht leer }              BEGIN
    loeschen;
    printkette,                                                     END;
END { kette }
```

Eine Ausführung dieses Programms kann beispielsweise so aussehen:

```
Gib Namen  A
Gib Namen  B
Gib Namen  C
Gib Namen  D
Gib Namen
Stand.  - A - B - C - D -
                                        zu loeschender Name: C
Stand:  - A - B - D -
                                        zu loeschender Name  A
Stand   - B - D -
                                        zu loeschender Name  D
Stand   - B -
                                        zu loeschender Name  X
*** X ist nicht in der Kette enthalten ***

Stand   - B -
                                        zu loeschender Name: B
Stand  Die Kette ist leer
```

PASCAL ist heute eine der beliebtesten Programmiersprachen der Welt. Die Kombination aus Mächtigkeit der Sprache und Einfachheit des Erlernens ist wohl einzig.

Dementsprechend ist PASCAL, der Nachkomme von ALGOL 60, selbst zum Ausgangspunkt neuer Entwicklungen geworden. Hier sind vor allem CONCURRENT PASCAL, MODULA und ADA zu nennen. Die Literatur zu PASCAL soll nur Beispiele nennen, es gibt sehr viel mehr Bücher zu diesem Thema [9, 11, 12, 13, 14, 15].

Strukturierte Programmierung

Das Thema «Strukturierte Programmierung» kann hier nur andeutungsweise behandelt werden, interessierte Leser seien auf die Literatur über Software Engineering [16] verwiesen.

In den sechziger Jahren entstanden erstmals große Programmsysteme, und es kam bald zu Problemen, die meist zu starken Überschreitungen der Termine und Kosten führten, oft sogar zum Scheitern großer Projekte. Als Gründe wurden vor allem die Schwierigkeiten ausgemacht, die sich durch die Komplexität der Programme ergeben. Als Gegenmittel wurde von DIJKSTRA die *strukturierte Programmierung* kreiert. Das Wort ist etwas schwammig, es wurde nie klar definiert. Sicher gehören dazu:

- Top-down-Programmentwicklung
- kleine, überschaubare Programmeinheiten, also wenige Zeilen (typisch als Prozeduren)
- pro Einheit nur ein einziger Ein- und Ausgang, also kein Einsprung mitten in Prozeduren oder Schleifen, keine alternativen Aussprünge aus Prozeduren
- keine Sprunganweisungen («GOTO considered harmful» von DIJKSTRA), denn diese erschweren das Verständnis des Programmablaufs (ein Programm mit vielen GOTOs gleicht einer Schale voll Spaghetti, in der man ja auch nicht leicht den Verlauf einer bestimmten Nudel verfolgen kann)
- keine Trickprogrammierung

Im weiteren Sinne gehören auch die sinnvolle Verwendung der Datentypen und eine gute, schritthaltend erstellte Dokumentation dazu.

Die wichtigsten Voraussetzungen für die Anwendung dieser Regeln sind

- die Schulung des Programmierers und
- die Bereitstellung von geeigneten Programmiersprachen

PASCAL bedeutet in diesem Sinne einen Meilenstein. Man beachte, daß bei allen Beispielen sehr kleine Einheiten entstanden waren, in denen nirgends eine Sprunganweisung vorkommt. Es wurde stets das Problem so programmiert, wie es ist, ohne daß für den Compiler irgendwelche Verrenkungen nötig waren. Darin liegt der erzieherische Einfluß der Programmiersprache: Der Programmierer soll sich auf das *Problem* konzentrieren, nicht auf einen problemfremden Lösungsmechanismus.

Natürlich ist es auch mit andern Sprachen möglich, den Prinzipien der strukturierten Programmierung treu zu bleiben. Aber es ist wie mit der Moral: Wer im Niederdorf wohnt, muß schon ein Heiliger sein, wenn er sich die Bilder nie ansieht. 13 TR

Literatur

9 Jensen K., Wirth N.: Pascal: user manual and report, Springer-Verlag, New York, 1978.
10 Wirth N.: Program development by stepwise refinement, Commun. ACM, Nr. 14, 1971.
11 Brown P. J.: PASCAL from BASIC, Addison-Wesley, Reading (Mass.), 1983.
12 Baumann R.: Programmieren mit PASCAL, Vogel Verlag, Würzburg, 1980.
13 Findlay W., Watt D. A.: PASCAL – An Introduction to Methodical Programming, Computer Science Press. Inc., Potomac (Md.), 1978.
14 Herschel R., Pieper F.: PASCAL – Systematische Darstellung von PASCAL und Concurrent PASCAL für den Anwender, Oldenburg Verlag, München, Wien, 1979.
15 Klein R. D.: Was ist PASCAL? Franzis Verlag, München, 1982.
16 Kimm R., Koch W., Simonsmeier W., Tontsch F.: Einführung in Software Engineering, de Gruyter Verlag, Berlin, New York, 1979.

Quelle: Der Beitrag ist aus einer Vortragsreihe in den «Technischen Abendkursen Baden» entstanden (9. November bis 21. Dezember 1983) und neu aufbereitet worden. Eine wesentlich erweiterte Fassung, in der zusätzlich die Sprache C und nichtkonventionelle Sprachen (LISP, LOGO, PROLOG, SMALLTALK) behandelt und Kapitel über Verifikation Struktogramme und Sprachklassifikation eingefügt sind, ist im Bibliographischen Institut in Mannheim erschienen.