[60] *Scott, E., Goetschel, D.:* One check bit per word can correct multibit errors. Electronics May 5 (1981), pp. 130–134.
[61] *Siegel, H. J., McMillen, R. J.:* The Cube Network as a Distributed Processing Test Bed Switch. Proc. 2nd Int. Conf. on Distr. Comp. Sys (1981).
[62] *Siegel, H. J., McMillen, R. J.:* Using the Augmented Data Manipulator Network in PASM. Computer 14, 2 (1981), pp. 25–33.
[63] *Siegel, H. J. et al.:* A survey of interconnection methods for reconfigurable parallel processing systems. Proc. NCC (1979), p. 529.
[64] *Siegel, H. J. et al.:* PASM: A Partionable SIMD/MIMD System for Image Processing and Pattern Recognition. Rep. School of El. Eng., Purdue Univ., W. Lafayette (1981) (not publ.).
[65] Siemens AG: PERKEO – A Hardware/Software System for „Personal Scientific Computing". Rep. Centr. Res. a Devel. Lab., München (1981) 1–30 (not publ.).
[66] *Snyder, L.:* Introduction to the Configurable, Highly Parallel Computer. Rep. CSD-TR-351, Purdue Univ., W. Lafayette (1981).
[67] *Snyder, L.:* Overview of the Chip Computer. Rep. CSD-TR-377, Purdue Univ., W. Lafayette (1981), pp. 1–10.
[68] *Snyder, L.:* Programming Processor Interconnection Structures. Rep. CSD-TR-381, Purdue Univ., W. Lafayette (1981), pp. 1–25.
[69] *Syiek, D. A., Agin, G. J.:* The CM-Z80 Multiprocessor Module. Rep. Robotics Inst., Carnegie Mellon Univ., Pittsburgh (1981) (not publ.).
[70] *Thurber, K. J.:* Parallel Processor Architectures – General Purpose Systems. Comp. Design, Jan. 1979, p. 89.
[71] *Thurber, K. J.:* Parallel Processor Architectures – Special Purpose Systems. Comp. Design, Feb. 1979, p. 103.
[72] U.S. NIM Commitee: FASTBUS Specifications. Sept. 1980.
[73] *Wall, E.:* Applying the Hamming Code to Microprocessor-based Systems. Electronics 22 (1979), p. 103.
[74] *Wiemers, M. et al.:* Ein schnelles Multitasking-Realtime-System zur simultanen Datenerfassung und Analyse von vielparametrigen Experimenten. MPI f. Kernphysik Heidelberg, V13 (1976).
[75] *Wulf, W. A., Harbison, S. P.:* Reflections in a Pool of Processors. Rep. CMU-CS-78-103, Carnegie Mellon Univ., Pittsburgh (1978), pp. 1–27.
[76] *Wulf, W. et al.:* HYDRA/C. mmp. New York (1981), pp. 1–297.
[77] Zilog: Zilog Z 8000 Family Technical Overview. No. 03-8074-01.
[78] Zilog: Z 8000 PLZ/ASM Assembly Language Programming Manual. No. 03-3055-01 (1979).

Ada: die neuen Konzepte

Ada: the new concepts

Elektron. Rechenanl. 24 (1982), H. 4, S. 175–186
Manuskripteingang: 27. Januar 1982

von J. F. H. WINKLER
Zentralbereich Technik, Siemens AG, München

Ada ist eine neue Höhere Programmiersprache, die in den letzten Jahren im Auftrage des US-Verteidigungsministeriums für Zwecke der Systemimplementierung entwickelt wurde. Die Sprache, die auf den wesentlichen Forschungsergebnissen der 70er Jahre auf dem Gebiet der Programmiersprachen beruht, ist jedoch auch für andere Bereiche der Programmierung (z. B. technisch-wissenschaftliche) gut geeignet. Der Aufsatz behandelt die Sprachelemente von Ada, die im Vergleich zu den derzeit verbreiteten Programmiersprachen neu sind. Es handelt sich dabei um das Paketkonzept (Programmodul), die getrennte aber kontrollierte Übersetzung von Programmbausteinen, parallele Prozesse, Programmschemata (Makromechanismus), die Schnittstelle zu anderen Systemkomponenten wie Übersetzer und Apparatur, die kontrollierte Genauigkeit von Zahlendarstellungen und die Programmierumgebung.

Ada is a new high order language that has been developed during the last few years on behalf of the US Department of Defense. Based on research in programming languages during the seventies it is intended to be used primarily as a system implementation language, but it is also suited to other areas of programming (e.g. for scientific and numerical applications). The paper deals with those elements of Ada which are new in comparison to most programming languages in use today. These elements are: the package concept (program module); the separate but controlled translation of program units; concurrent processes; generic program units (macros); the interface to other system components such as the translator and the hardware; the controlled precision of number representations; and the programming environment.

1 Einleitung

Ada ist eine neue Programmiersprache, die im Auftrag des amerikanischen Verteidigungsministeriums in den letzten Jahren im Rahmen eines großangelegten Projektes entwickelt wurde. Die Definition der Sprache erfolgte durch eine Gruppe unter der Leitung von *J. Ichbiah* bei Cii-Honeywell Bull in der Zeit von 1977 bis 1980. Für die genaue Entstehungsgeschichte sei auf [1; 2; 3; 4; 5] verwiesen.

Nach PL/1 und Algol68 ist Ada der dritte große Versuch, eine möglichst universelle Programmiersprache zu schaffen. Es wurde zwar als Systemimplementierungssprache geplant, entworfen und realisiert, aber es ist auch für die technisch-wissenschaftliche Programmierung gut geeignet. Numerische Algorithmen lassen sich infolge der Möglichkeiten zur Spezifikation von Genauigkeiten der Zahlendarstellungen in Ada am ehesten portabel formulieren. Auch für die Programmierung kommerzieller Programmsysteme ist Ada geeignet, da die Eigenschaften und Ausdrucksweisen, die an die kaufmännische und betriebswirtschaftliche Problemwelt angepaßt sind, in Zukunft eher an der Mensch/Maschine-Schnittstelle inter-

aktiver Systeme verlangt werden als bei der Konstruktion und Programmierung dieser Systeme selbst, die mehr und mehr von Informatikern und Technikern durchgeführt werden wird.

Die vorliegende Arbeit will nur die wesentlichen und im Vergleich zu den meisten verbreiteten Programmiersprachen neuen Sprachelemente von Ada darstellen; eine Behandlung der gesamten Sprache ist schon wegen des beschränkten Umfangs hier nicht möglich.

Im Vergleich zu Pascal [6; 7] läßt sich der Sprachumfang von Ada grob durch folgende "Gleichung" charakterisieren:

Ada = Pascal
- – formale Prozeduren
- – set- und file-Typen
- – with-Anweisung
- + Paket
- + getrennte Übersetzung
- + Trennung von Spezifikation und Implementierung
- + Makromechanismus
- + Block
- + exit- und return-Anweisung
- + dynamische Datentypen (z. B. Reihungen)
- + abgeleitete Datentypen
- + Überladen von Unterprogrammnamen
- + Aggregate zur Notierung von Reihungs- und Verbundkonstanten
- + kontrollierte Genauigkeit von rationalen Zahlen
- + Attribute und Pragmata
- + statische Ausdrücke
- + parallele Prozesse
- + Ausnahmebehandlung
- + Festlegung der Datendarstellung im Speicher
- + Maschinencodeeinfügungen
- + sequentielle und wahlfreie Ein-/Ausgabe

Die Sprache Ada ist eine umfangreiche Sprache, die im Detail manche Sonderfälle aufweist. Aus diesem Grund wird im folgenden nur auf die grundlegenden Eigenschaften der betrachteten Sprachelemente eingegangen. Für weitere Einzelheiten sei auf die Sprachdefinition [8] und einführende Literatur verwiesen [9; 10; 11; 12; 53; 54]. Eine ausführliche Bibliographie über Ada wurde veröffentlicht in: Ada Letters I, 1 (1981), S. 50–63, I, 2 (1981/82), S. 41–43, I, 3 (1982), S. 93–94, I, 4 (1982), S. 77–78. Die derzeit gültige Sprachbeschreibung wird Mitte 1982 durch eine in einigen Punkten geänderte Fassung ersetzt werden. Die Änderungen sind in [13] beschrieben. Sie betreffen in der Regel Detailfragen, so daß der Inhalt dieses Aufsatzes davon nicht berührt wird. Im Zweifelsfalle erfolgt die Darstellung nach [13].

Im folgenden zweiten Abschnitt wird die Struktur von Ada-Programmen im Großen behandelt. Ada ermöglicht die Formulierung von Programmen, die aus getrennt übersetzbaren Einheiten bestehen, wobei jedoch dieselbe semantische Konsistenz wie innerhalb einer Übersetzungseinheit gewährleistet wird. Die Grobstruktur der Programme ist außerdem gekennzeichnet durch die Trennung von Spezifikation (Schnittstelle) und Implementierung.

Mit der Schnittstelle der Ada-Programme zu ihrer jeweiligen Umgebung befaßt sich der dritte Abschnitt. Ada bietet mehrere Möglichkeiten, Beziehungen zwischen dem Programm und seiner Umgebung auszudrücken.

Im vierten Abschnitt werden die numerischen Sprachelemente von Ada behandelt, wobei die Betonung auf dem Aspekt der Portabilität liegt.

Ada enthält auch parallele Prozesse, wobei hier für die Synchronisation und Kommunikation neuere Ansätze von *Hoare* [14] und *Brinch-Hansen* [15] realisiert sind. Darauf wird im fünften Abschnitt eingegangen.

Der sechste Abschnitt befaßt sich mit den Unterprogramm- und Paketschemata (generics), die einen auf Unterprogramme und Pakete eingeschränkten Makromechanismus darstellen.

Abschließend wird im Abschnitt 7 auf die für Ada vorgesehene Programmierumgebung (APSE) eingegangen. Da sich diese Programmierumgebung noch in einer sehr frühen Phase der Entwicklung befindet, kann hier nur auf die Motivation und die wesentlichen Ideen eingegangen werden.

2 Programmstruktur

2.1 Allgemeines

Gegenüber den meisten bisherigen Sprachen bietet Ada auch Sprachelemente zum Programmieren im Großen ("Makroprogrammierung") und erfüllt damit eine Forderung, die zumindest seit dem Aufsatz von *deRemer* und *Kron* [16] allgemein bewußt war und einer Lösung harrte. Wesentlich für die Grobstruktur von Ada-Programmen sind die separate Übersetzbarkeit, die Bildung von Paketen (Module) und die Trennung von Spezifikation und Implementierung. Diese drei Aspekte sollen im folgenden behandelt werden.

2.2 Pakete

Ada enthält das Konzept des Programmoduls im Sinne von [17; 18; 19]. Die Module, die in Ada Pakete genannt werden, sind nach folgendem Schema aufgebaut:

package BEZEICHNER is VEREINBARUNGSFOLGE_1 [private VEREINBARUNGSFOLGE_2] end [BEZEICHNER];	Spezifikation
package body BEZEICHNER is [VEREINBARUNGSTEIL] [begin ANWEISUNGSFOLGE [exception AUSNAHMETEIL]] end [BEZEICHNER];	Implementierung

Ein Paket ist allgemein eine Zusammenfassung von Vereinbarungen (z. B. von Typen, Konstanten, Variablen, Unterprogrammen, Prozessen, Paketen). Das einfachste Paket besteht nur aus einem Spezifikationsteil ohne private-Teil und enthält dann nur Datenvereinbarungen (z. B. Typen, Objekte). Ein solches Paket kann man als reines Datenpaket bezeichnen. Ein komplizierteres Paket, welches einen private-Teil und einen Implementierungsteil enthält, könnte z. B. ein Paket sein, welches eine Datenbank realisiert.

Von außen angesprochen werden können höchstens der BEZEICHNER und die Größen, die in der VEREINBARUNGSFOLGE_1 (öffentlicher Teil) vereinbart sind. Der Teil zwischen private und end (privater Teil) dient dem Verbergen modulinterner Festlegungen, z. B. bei der Realisierung abstrakter Datentypen. Diese Festlegungen im privaten Teil sind für den Übersetzer bestimmt, um die getrennte Übersetzung von Spezifikation und Implementierung realisieren zu können.

Zur Realisierung abstrakter Datentypen können in der VEREINBARUNGSFOLGE_1 Datentypen der Art private oder limited private vereinbart werden. Bei private sind für Werte dieser Typen nur die Operationen ":=", "=" und "/=" standardmäßig verfügbar, bei limited private dagegen überhaupt keine. In der VEREINBARUNGSFOLGE_1 können nun Funktionen und Prozeduren vereinbart werden, die solche Typen als Parameter- und/oder als Ergebnistyp haben. Solche Unterprogramme (und bei limited private nur solche) können dann von außen zur Manipulation von Werten solcher privaten Datentypen verwendet werden. Im privaten Teil wird dann festgelegt, wie private Typen auf bereits vereinbarte Typen zurückgeführt werden.

Die Realisierung abstrakter Datentypen in Ada soll im folgenden Beispiel 2–1 von Mengen ganzer Zahlen gezeigt werden.

```
package GZ_Mengen_Paket is
  type GZ_Menge is limited private;
  Leere_GZ_Menge: constant GZ_Menge;
  procedure Einfuegen (Elem: in INTEGER;
                       Menge: in out GZ_Menge);
  function "/" (Elem: in INTEGER;
                Menge: in GZ_Menge)
               return BOOLEAN; -- Element Test
private
-- GZ_Menge als verkettete Liste realisiert
   type Mengen_Element;
   type GZ_Menge is access Mengen_Element;
   type Mengen_Element is
     record Element: INTEGER;
       Nachfolger: GZ_Menge;
     end record;
   Leere_GZ_Menge: constant GZ_Menge:=null;
end GZ_Mengen_Paket;

package body GZ_Mengen_Paket is
  procedure Einfuegen (Elem: in INTEGER;
                       Menge: in out GZ_Menge) is
  begin
    if not (Elem/Menge)
    then Menge:=new Mengen_Element (Elem,
                                    Menge);
    end if;
  end Einfuegen;

  function "/" (Elem: in INTEGER;
                Menge: in GZ_Menge)
               return BOOLEAN is
  begin
    if Menge=null
    then return FALSE;
    elsif Menge. Element=Elem
    then return TRUE;                          (2–1)
    else return Elem/(Menge. Nachfolger);
    end if;
  end "/";
end GZ_Mengen_Paket;
```

Das Paket "GZ_Mengen_Paket" exportiert nach außen den Typ "GZ_Menge", die Konstante "Leere_GZ_Menge", die Prozedur "Einfuegen" und die Funktion "/". Der Typ "Mengen_Element", der bei der Realisierung von "GZ_Menge" verwendet wird, kann außerhalb des Paketes nicht verwendet werden.

Der Spezifikationsteil des Paketes enthält für die öffentlichen Prozeduren und Funktionen nur die Spezifikation derselben, d. h. die entsprechenden Köpfe. Nur diese sind für die Benutzung von außen notwendig. Die Rümpfe dieser Unterprogramme sind im Paketrumpf enthalten, und zwar im VEREINBARUNGSTEIL. Der Paketrumpf und damit z. B. auch die Rümpfe dieser Unterprogramme können beliebig modifiziert werden, ohne daß ein Benutzer des Paketes davon berührt wird. Nur wenn der Spezifikationsteil in seinem öffentlichen Teil geändert wird, können Benutzer des Paketes betroffen sein und müssen dann ihrerseits Programmodifikationen vornehmen. Dies gilt nur im rein formalen Sinne und nicht inhaltlich, da z. B. die Wirkung (Semantik) der im Spezifikationsteil vereinbarten Unterprogramme dort nicht festgelegt werden kann.

Der Einfachheit wegen sind in dem o. a. Paket nur die zwei Operationen "Einfuegen" und "/" definiert worden. Weitere Mengenoperationen könnten analog dazu definiert werden.

Das Mengenpaket kann folgendermaßen angewendet werden:

```
Vereinbarung von Mengenvariablen:
  M1, M2, M3: GZ_Menge;
Manipulationen:
  Einfuegen (10, M1);
  Einfuegen (25, M1);
  if 23/M1
  then Einfuegen (24, M1);
  end if;
```

2.3 Trennung von Spezifikation und Implementierung

Im Sinne der Bildung klarer Schnittstellen zwischen Programmeinheiten ermöglicht Ada die Trennung von Spezifikation (=Schnittstelle zur benutzenden Umgebung) und Implementierung (=Realisierung) für Pakete, Unterprogramme, Prozesse und einige Datentypen. Die Realisierung von Datentypen wird in Abschnitt 3 behandelt.

Für Pakete wurde diese Trennung von Spezifikation und Implementierung bereits in Abschnitt 2.2 dargestellt, so daß hier nicht weiter darauf eingegangen wird.

Für Unterprogramme besteht die Spezifikation aus dem Unterprogrammkopf, die Implementierung aus der vollständigen Unterprogrammvereinbarung, d.h. aus Kopf und Rumpf. Auch dafür wurde bereits in Abschnitt 2.2 ein Beispiel gegeben, und zwar durch die Unterprogramme "Einfuegen" und "/". Deren Köpfe sind im Spezifikationsteil des Paketes "GZ_Mengen_Paket" notiert, während die Rümpfe im zugehörigen Implementierungsteil aufgeführt sind. Bei Unterprogrammen wird im Implementierungsteil die Spezifikation wiederholt. Dies erhöht vor allem die Lesbarkeit. Die Erfahrung zeigt, daß z. B. sorgfältige Pascal-Programmierer bei Unterprogrammen, die zuerst als FORWARD vereinbart sind, in der endgültigen Vereinbarung die Parameterliste, die an dieser Stelle in Pascal nicht mehr zugelassen ist [7; 6], in Kommentarform wiederholen [20].

Bei Prozessen sind Spezifikations- und Implementierungsteil nach folgendem Schema aufgebaut:

```
task BEZEICHNER [is                          | Spezifika-
   ENTRY_VEREINBARUNGEN                      | tion
   REPRÄSENTATIONEN                          |
end [BEZEICHNER]];                           |

task body BEZEICHNER is                      |
   [VEREINBARUNGSTEIL]                       |
begin                                        | Imple-
   ANWEISUNGSFOLGE                           | mentie-
[exception                                   | rung
   AUSNAHMETEIL]                             |
end [BEZEICHNER];                            |
```

Auf die Bedeutung der ENTRY_VEREINBARUNGEN, welche die wesentlichen Teile der Prozeßspezifikation sind, wird in Abschnitt 5 eingegangen.

Die Trennung von Spezifikation und Implementierung von Unterprogrammen, Paketen und Prozessen ist auch innerhalb eines VEREINBARUNGSTEIL möglich. Dadurch ist die Vereinbarung sich gegenseitig referierender Programmeinheiten generell ohne besondere Hilfsmittel möglich. Der größte Gewinn der Trennung von Spezifikation und Implementierung ergibt sich im Zusammenhang mit der getrennten Übersetzung von Programmeinheiten. Darauf wird im folgenden Abschnitt näher eingegangen.

2.4 Getrennte Übersetzung und Bibliothek

Ein Ada-Hauptprogramm ist stets eingebettet in eine Ada-Programmbibliothek. Diese Bibliothek besteht aus einer Menge getrennt übersetzbarer Übersetzungseinheiten, zwischen denen bestimmte Relationen bestehen.

Unter strukturellen Gesichtspunkten gibt es zwei Arten von Übersetzungseinheiten:

- Bibliothekseinheiten
- Untereinheiten.

Untereinheiten sind dadurch ausgezeichnet, daß sie, obwohl getrennt übersetzbar, im Sinne der statischen Programmstruktur im Innern einer anderen Übersetzungseinheit liegen. Bibliothekseinheiten sind alle Übersetzungseinheiten, die keine Untereinheiten sind.

2.4.1 Bibliothekseinheiten

Bibliothekseinheiten können sein:

- Spezifikationsteil von Unterprogrammen
- Implementierungsteil von Unterprogrammen
- Spezifikationsteil von Paketen
- Implementierungsteil von Paketen

Durch die getrennte Übersetzung von Spezifikation und Implementierung wird errreicht, daß die Implementierung modifiziert (z. B. verbessert) werden kann, ohne daß die Benutzer davon betroffen werden, denn die Schnittstelle nach außen ist nur die Spezifikation (s. Abschn. 2.2), die stets vor der zugehörigen Implementierung (Rumpf) übersetzt werden muß (SI-Relation).

Wenn keine weitere Maßnahmen ergriffen werden, sind getrennt übersetzte Bibliothekseinheiten voneinander isoliert, außer daß der übliche Zusammenhang zwischen zusammengehörigem Spezifikations- und Implementierungsteil besteht. Eine Bezugnahme zwischen Bibliothekseinheiten ist darüber hinaus mit der with-Klausel möglich. Dies wird im folgenden Programm 2–2 gezeigt:

```
with GZ_Mengen_Paket;
procedure P is
   Menge: GZ_Mengen_Paket. GZ_Menge;
begin                                                    (2–2)
   ...
   GZ_Mengen_Paket.Einfuegen(1981,Menge);
   ...
end P;
```

In einer Übersetzungseinheit A, die in ihrer with-Klausel die Übersetzungseinheit B aufführt, können alle Größen C des öffentlichen Teils von B in der Form B.C referiert werden.

Das obige Beispiel zeigt, daß die Verwendung von mnemotechnisch guten Bezeichnern zu einem recht hohen Schreibaufwand führen kann und daß auch die Lesbarkeit darunter leiden kann. Andererseits ist bekannt, daß die Verständlichkeit von Programmen verbessert wird, wenn in einem beschränkten und überschaubaren Be-

reich Abkürzungen verwendet werden. Dazu gibt es in Ada zwei Möglichkeiten: die use- und die renames-Klauseln, deren Wirkung in den Varianten der Prozedur P in den Programmen 2–3 und 2–4 gezeigt ist.

```
with GZ_Mengen_Paket;
procedure P is
  use GZ_Mengen_Paket;
  Menge: GZ_Menge;
begin
   ...                                          (2–3)
  Einfuegen (1981, Menge);
   ...
end P;
```

Die Wirkung der use-Klausel, in welcher nur Pakete genannt werden können, besteht in einem Herausfaktorisieren der darin genannten Bezeichner analog zur Wirkung der with-Klausel in Pascal [7].

```
with GZ_Mengen_Paket;
procedure P is
  package GZM renames GZ_Mengen_Paket;
  Menge: GZM.GZ_Menge;
begin                                           (2–4)
   ...
  GZM.Einfuegen(1981,Menge);
   ...
end P;
```

Die renames-Klausel erlaubt eine Umbenennung und damit auch Abkürzung von anderswo vereinbarten Größen.

Durch die with-Klausel wird zwischen Übersetzungseinheiten eine asymmetrische Relation definiert (with-Relation). Diese muß darüber hinaus noch zykelfrei sein. Diese Vorschrift folgt aus dem in Ada allgemein geltenden "linear text model" ("zuerst vereinbaren, dann verwenden"). Wenn in Einheit A die Klausel "with B;" enthalten ist, dann muß B vor A übersetzt werden.

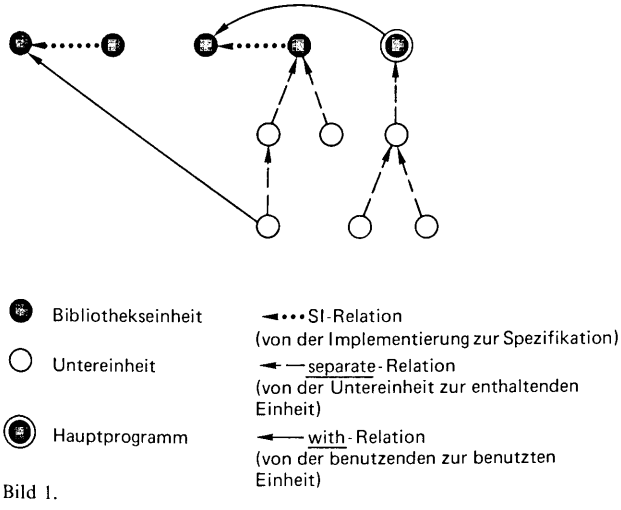

Bild 1.

2.4.2 Untereinheiten

Als Untereinheiten können folgende Größen auftreten:

- Implementierungsteil eines Paketes
- Implementierungsteil eines Unterprogrammes
- Implementierungsteil eines Prozesses

Die Vereinbarung einer Untereinheit erfolgt in einer anderen Übersetzungseinheit, und zwar durch einen entsprechenden Spezifikationsteil und einen sogenannten Implementierungsstumpf. Dies ist im Programm 2–5 an einer alternativen Formulierung des Rumpfes von GZ_Mengen_Paket (s. Programm 2–1) gezeigt.

```
package body GZ_Mengen_Paket is
  procedure Einfuegen is
    -- wie im Programm 2–1
  end Einfuegen;
  function "/" (Elem: in INTEGER;
                Menge: in GZ_Menge)
               return BOOLEAN is
               separate; -- Implementierungsstumpf
end GZ_Mengen_Paket;                            (2–5)
separate (GZ_Mengen_Paket ) -- Beginn der
                            -- Untereinheit
function "/" (Elem: in INTEGER;
              Menge: in GZ_Menge)
             return BOOLEAN is
  Hz: GZ_Menge:=Menge;
begin
  while Hz /= null
  loop if Hz.Element=Elem
       then return TRUE;
       else Hz := Hz.Nachfolger;
       end if;
  end loop;
  return FALSE;
end "/";
```

Statt des Paketrumpfes in Programm 2–1 könnte man auch den Rumpf (incl. dem separaten Rumpf von "/") aus Programm 2–5 verwenden, ohne die Spezifikation im Programm 2–1 ändern oder neu übersetzen zu müssen. Solange die Schnittstellen eingehalten werden, dürfen beliebig neue Implementierungsteile in das Programm eingebracht werden. Im Programm 2–5 kann man außerdem den Rumpf von "/" durch einen anderen ersetzen, ohne Spezifikation und/oder Rumpf von GZ_Mengen_Paket ändern oder neu übersetzen zu müssen.

Innerhalb einer Untereinheit besteht dieselbe Sicht wie an der Stelle des Implementierungsstumpfes. Daraus folgt, daß die Einheit, die den Implementierungsstumpf enthält, vor der Untereinheit übersetzt werden muß. Innerhalb einer Untereinheit kann wieder ein Implementierungsstumpf auftreten, d.h., es können beliebige Hierarchien von Untereinheiten gebildet werden (top–down Entwurf). Durch die Bildung von Untereinheiten wird eine dritte Ordnungsrelation zwischen Übersetzungseinheiten definiert: die separate-Relation.

2.4.3 Bibliothekstruktur und Hauptprogramm

Insgesamt läßt sich damit die Struktur einer Ada-Bibliothek folgendermaßen charakterisieren (s. Bild 1):

a) die Menge der Bibliothekseinheiten ist halbgeordnet (with-Relation, SI-Relation);
b) eine Bibliothekseinheit kann Wurzel eines Baumes sein, dessen übrige Knoten Untereinheiten sind;
c) zwischen einer Untereinheit und einer Bibliothekseinheit kann die with-Relation bestehen.

Eine Ada-Bibliothek ist konsistent, wenn der durch a), b) und c) definierte Graph zykelfrei ist und zu jeder Spezifikation, die kein reines Datenpaket ist, eine vollständige Implementierung (also nicht nur ein Stumpf) vorhanden ist. Die Übersetzungseinheiten einer Bibliothek müssen in einer Reihenfolge übersetzt werden, die mit den drei genannten Ordnungsrelationen verträglich ist. In Bild 1 ist ein Beispiel für eine Ada-Bibliothek angegeben.

Abschließend soll nun darauf eingegangen werden, was ein Ada-Hauptprogramm ist und wie es aus Bibliotheks- und Untereinheiten aufgebaut sein kann.

Eine Bibliothekseinheit, die ein Unterprogramm ist, kann als Hauptprogramm ausgezeichnet werden. Vor dem Start dieses Hauptprogrammes müssen alle Bibliothekseinheiten, die über with-Klauseln vom Hauptprogramm oder von erreichbaren Untereinheiten erreichbar sind, abgearbeitet werden, wobei "erreichbar" im transitiven Sinne zu verstehen ist. Daran anschließend wird dann das ausgezeichnete Unterprogramm abgearbeitet.

3 Schnittstelle zur Umgebung

Ein Ada-Programm wird im allg. übersetzt und ausgeführt. Während dieser beiden Verarbeitungsschritte kommt es mit mehreren anderen Systemkomponenten in Berührung: z. B. Übersetzer, Apparatur, Laufzeitsystem, Betriebssystem. Ada enthält eine Reihe von Elementen zur Kommunikation mit diesen Systemkomponenten.

3.1 Pragmas

Pragmas ermöglichen die Steuerung des Übersetzungsprozesses. Es können z. B. Vorgaben für die Art der Protokollierung, der Optimierung und der Strategie zur Codeerzeugung gemacht werden. Syntaktisch haben Pragmas im wesentlichen die Form von Prozeduraufrufen, denen pragma vorangestellt ist, z. B.:

```
pragma OPTIMIZE(TIME);
```

Dieser Hinweis teilt dem Übersetzer mit, daß ein möglichst schnelles Objektprogramm erzeugt werden soll und der Speicherbedarf desselben weniger wichtig ist.

```
pragma INTERFACE (FORTRAN,
                  Matrix_Inversion);
```

Dieser Hinweis teilt dem Übersetzer mit, daß der Rumpf des Unterprogramms "Matrix_Inversion" in FORTRAN formuliert ist. Für solche Unterprogramme wird im Ada-Programm lediglich der Programmkopf, d.h. die Schnittstelle, spezifiziert.

Außer den in der Sprache festgelegten Pragmas können in einer Implementierung von Ada weitere Pragmas definiert werden. Um die Übertragbarkeit dadurch nicht zu beeinträchtigen, ist festgelegt, daß ein unbekanntes Pragma keinerlei Wirkung hat.

In bisherigen Programmiersprachen waren solche Hilfsmittel in der Regel nicht enthalten. Sie waren dann stets implementierungsspezifisch, wodurch die Übertragbarkeit oft beeinträchtigt wurde, wenn für diese Hilfsmittel nicht die syntaktische Form von Kommentaren gewählt wurde.

3.2 Attribute

Attribute sind Sprachmittel zum Festlegen und Abfragen von Eigenschaften und Kenngrößen von Objekten, Typen, Unterprogrammen und Nachrichteneingängen von Prozessen. Dabei handelt es sich sowohl um Eigenschaften auf Quellebene als auch um solche im Objektprogramm. Die ersteren dienen vor allem dazu, eine größere Kohärenz des Quellprogramms zu ermöglichen. Beispiele dafür sind:

FIRST(J): gibt die untere Grenze des J-ten Index einer Reihung oder eines Reihungstyps an.

PRED: liefert für alle diskreten Datentypen den Vorgänger eines gegebenen Wertes, sofern dieser Vorgänger existiert.

Die Kohärenz des Quellprogrammes wird nun dadurch erreicht, daß z. B. für eine zweidimensionale Reihung K_Matrix der Ausdruck "K_Matrix'FIRST(2)" stets den Wert der unteren Grenze der zweiten Dimension von K_Matrix liefert, d.h., bei einfachen Änderungen der Vereinbarung von K_Matrix bleibt der Ausdruck "K_Matrix'FIRST(2)" gültig und muß seinerseits nicht geändert werden.

Die auf das Objektprogramm bezogenen Attribute hängen damit zusammen, daß Ada als Systemimplementierungssprache die Ansprache und Benutzung aller Apparatureigenschaften erlauben soll. Beispiele dafür sind:

ADDRESS: gibt die Adresse der ersten von einem Objekt oder Unterprogramm belegten Speichereinheit an.

SIZE: gibt die maximale Anzahl von Bits an, die Objekte eines bestimmten Typs (ungleich Tasktyp) benötigen.

3.3 Festlegung von Speicherabbildern

Da Ada eine Systemimplementierungssprache ist, enthält es auch eine Reihe von Sprachelementen, die es gestatten, Einfluß auf die Realisierung von im Programm vereinbarten Größen im Speicher zu nehmen. Dafür gibt es folgende Hilfsmittel:

- Festlegung der Adresse eines Datenobjektes.

- Festlegung der maximalen Anzahl von Bits, die zur Darstellung von Objekten eines gegebenen Typs verwendet werden können.
- Festlegung des maximalen Speicherbereiches, der zum Abspeichern von Objekten verwendet werden kann, auf die Verweise (pointer) eines bestimmten Verweistyps zeigen.
- Festlegung des für einen Prozeß oder Objekte eines Prozeß-Typs verfügbaren Datenspeicherbereiches.
- Festlegung der Darstellung der Werte von Aufzählungsdatentypen. Hierbei ist die beliebige Zuordnung ganzer Zahlen zulässig, solange die Zuordnung ordnungstreu ist.
- Festlegung der Darstellung von Verbundobjekten. Hierbei ist es möglich anzugeben, an welchen Speicherstellen solch ein Objekt liegen darf (alignment) und wo die einzelnen Komponenten innerhalb des Objektes liegen.
- Zuordnung von Unterbrechungen zu Nachrichteneingängen von Prozessen.

Zusammen mit der Möglichkeit, in sogenannten Code-Prozeduren direkt den Code des Objektprogrammes anzugeben, lassen sich in Ada alle Eigenschaften auf der Maschinenbefehlsebene von Rechnern ansprechen. Der völlig freizügige Zugang zum Speicher, wie er z.B. zur Realisierung einer Speicherverwaltung benötigt wird, ist durch eine Typkonversion ohne Überprüfung der Kompatibilität (UNCHECKED_CONVERSION) möglich.

4 Genauigkeit von Zahlen

Um die Übertragbarkeit numerischer Programme zu gewährleisten, enthält Ada Sprachelemente zur Festlegung der Genauigkeit von Zahlendarstellungen. Diese Sprachelemente sind in das Konzept der Datentypen eingefügt. Für eine ausführliche Darstellung siehe [21].

Bei den ganzen Zahlen gibt es wie in Pascal die Möglichkeit, als Wertebereich eines Ganzzahltyps ein Intervall anzugeben. Bei den nichtganzen Zahlen werden Fest- und Gleitpunkttypen unterschieden, wobei ein Datentyp allgemein eine Menge von Werten (und eine Menge von Operationen) ist.

4.1 Festpunkttypen

Festpunkttypen sind dadurch ausgezeichnet, daß die Abweichung zwischen extern notierter Zahl und interner Repräsentation für alle Werte des Typs durch eine feste Schranke nach oben beschränkt ist. Ein Festpunkttyp wird durch drei Größen definiert, die in der Typdefinition in der Schreibweise (4–1) angegeben werden.

```
delta STAT_AUSDR_1 range
STAT_AUSDR_2..STAT_AUSDR_3          (4–1)
```

Der Ausdruck nach dem Schlüsselwort delta ist die oben erwähnte Schranke, die stets positiv sein muß. Der durch (4–1) definierte Festpunkttyp wird intern repräsentiert durch eine Zahlenmenge der Form (4–2).

$$d^*(-(2^{**}N)+1..(2^{**}N)-1) \qquad (4\text{–}2)$$

Dabei ist $0 < d \leq$ STAT_AUSDR_1 und N eine positive, ganze Zahl so, daß STAT_AUSDR_2 und STAT_AUSDR_3 höchstens um STAT_AUSDR_1 von Zahlen der Menge (4–2) entfernt liegen. Die Zahlen der Menge (4–2) werden als Modellzahlen des durch (4–1) definierten Typs bezeichnet.

4.2 Gleitpunkttypen

Bei Gleitpunkttypen hängt die Abweichung zwischen extern notierter Zahl und interner Repräsentation vom Absolutwert der betreffenden Zahl ab. Es handelt sich bei dieser Abweichung also um einen relativen Fehler.

Ein Gleitpunkttyp wird durch drei Größen charakterisiert, die in der Typdefinition in der Schreibweise (4–3) angegeben werden.

```
digits STAT_AUSDR_1 range
STAT_AUSDR_2..STAT_AUSDR_3          (4–3)
```

Der statische, d.h. während der Übersetzung berechenbare Ausdruck STAT_AUSDR_1, dessen Wert eine positive ganze Zahl sein muß, gibt die Anzahl der signifikanten Dezimalziffern der Mantisse der exakt repräsentierten Zahlen an. Die Modellzahlen des durch (4–3) definierten Gleitpunkttyps sind in (4–4) angegeben. Q bedeutet die Menge der rationalen und Z die der ganzen Zahlen.

$$\{0\} \cup \{x \in Q \mid x = \pm \text{bm} * (2^{**}\text{exp}) \wedge -4*B \leq \text{exp} \leq 4*B \wedge \text{exp} \in Z \wedge B = \lceil \text{STAT\_AUSDR\_1} * \log(10)/\log(2) \rceil\}$$

Die Mantisse bm hat genau B Stellen nach dem Punkt, wenn sie binär ausgedrückt wird.

Verknüpfungen und Vergleiche zwischen rationalen Zahlen sind definiert durch entsprechende Verknüpfungen und Vergleiche zwischen den kleinsten Intervallen, welche als Grenzen Modellzahlen haben und den jeweiligen Operanden enthalten (Modellintervalle). Jede gültige Implementierung muß diese mathematischen Eigenschaften gewährleisten, auch wenn sie keine Intervallarithmetik verwendet.

5 Parallele Prozesse

Beim Prozeßkonzept geht Ada im Vergleich zu früheren Programmiersprachen neue Wege, indem es für Synchronisation und Kommunikation neuere Ideen und Vorschläge von *Brinch-Hansen* und *Hoare* [15; 14] aufgreift.

Der Prozeßbegriff von Ada deckt sich mit dem in [22; 23] definierten Prozeßbegriff, wo unter einem Prozeß im wesentlichen die einmalige Ausführung einer Prozedur verstanden wird.

In Ada werden Prozesse als Objekte der Art task defi-

niert. Wie andere Programmeinheiten bestehen task-Objekte aus einer Spezifikation und einer Implementierung. Der grobe Aufbau beider wurde bereits in Abschnitt 2.3 angegeben.

Beginnt die Spezifikation mit task type, dann wird ein Prozeßtyp definiert, der genauso verwendet werden kann wie ein Datentyp der Art limited private (s. Abschn. 2.2); insbesondere zur Vereinbarung von Objekten, in der Definition zusammengesetzter Datentypen und als Typ von formalen Parametern.

Jeder durch Abarbeitung einer Vereinbarung oder durch Ausführung von new erzeugte Prozeß wird genau einmal ausgeführt.

5.1 Synchronisation und Kommunikation

Als Grundelement von Synchronisation und Kommunikation wird der synchrone Nachrichtenaustausch in Form eines "einfachen Gesprächs" (Rendezvous in [8]) verwendet:

A → B (A spricht B an: Parameterübergabe)
B (B legt die Antwort fest: Ausführung des Rumpfes)
A ← B (B antwortet A: Parameterrückgabe)

Der Nachrichtenaustausch erfolgt über Nachrichteneingänge, die in der Spezifikation eines Prozesses vereinbart werden können. Konzeptionell entsprechen diese Nachrichteneingänge den "ports" in [24], den Nachrichteneingängen in [25] oder den "common procedures" in [15]. Der Aufbau der Vereinbarung eines Nachrichteneinganges ist in (5–1) angegeben.

```
entry EING_BEZ [(DISKR_BEREICH)]
[FORM_PAR_LISTE];                    (5–1)
```

Die Angabe von DISKR_BEREICH ermöglicht die Vereinbarung einer Menge von Nachrichteneingängen, die im wesentlichen einer eindimensionalen Reihung mit dem durch DISKR_BEREICH gegebenen Indexbereich entspricht. Durch FORM_PAR_LISTE kann wie bei einem Unterprogramm eine Liste formaler Parameter angegeben werden. Nach außen sieht daher ein Nachrichteneingang im wesentlichen wie eine Prozedur aus.

Ein Gespräch (Rendezvous) findet zwischen zwei verschiedenen Prozessen statt, dem Rufer und dem Gerufenen. Der Rufer ruft mit einer Anweisung der Form (5–2) einen Nachrichteneingang auf.

```
PROZESS_NAME.EING_NAME
[(AKT_PARAM_LISTE)];                 (5–2)
```

Der gerufene Prozeß PROZESS_NAME führt eine Annahmeanweisung der Form (5–3) aus.

```
accept EING_NAME [FORM_PARAM_LISTE]
[do ANWEISUNGSFOLGE                  (5–3)
end [EING_BEZ]];
```

Die Annahmeanweisung (5–3) entspricht ungefähr einem Prozedurrumpf: der Name des Nachrichteneingangs und die Liste der formalen Parameter werden wiederholt und als Rumpf eine Anweisungsfolge hinzugefügt. Ein Unterschied zu Prozeduren besteht darin, daß es innerhalb des gerufenen Prozesses zu einem Nachrichteneingang beliebig viele Annahmeanweisungen geben kann, die auch geschachtelt sein können.

Da in der Annahmeanweisung im Gegensatz zum Aufruf kein Prozeß genannt wird, kann das Gespräch als einseitig anonym bezeichnet werden.

Das Gespräch besteht aus zwei Phasen: Synchronisations- und Kommunikationsphase. In der Synchronisationsphase wird gewartet, bis die Bedingung

"Aufruf ausgeführt und Annahmeanweisung erreicht" (5–4)

erfüllt ist. Wird eine Teilbedingung früher als die andere erfüllt, dann wartet der betreffende Prozeß auf den Gesprächspartner. Muß der Rufer warten, dann wird er in eine dem angesprochenen Nachrichteneingang zugeordnete Warteschlange eingereiht und in einen Wartezustand versetzt. Führt ein Prozeß eine Annahmeanweisung für einen Nachrichteneingang bei leerer Warteschlange aus, dann muß er warten, bis die Bedingung (5–4) erfüllt wird.

Die Kommunikationsphase besteht aus der Parameterübergabe, der einmaligen Ausführung der ANWEISUNGSFOLGE und der Parameterrückgabe. Dadurch wird genau ein Aufruf befriedigt. Die Abarbeitung des rufenden Prozesses wird erst nach der Beendigung der Kommunikationsphase fortgesetzt: es handelt sich um eine synchrone Kommunikation. Warten mehrere rufende Prozesse in der Warteschlange eines Nachrichteneingangs, dann wird in einem Gespräch der Aufrufer befriedigt, der als erster eingereiht wurde. Nach dem Gespräch laufen beide Prozesse unabhängig voneinander weiter.

Die eben dargestellte einfachste Form des Gesprächs kann sowohl auf der Seite des Rufers als auch des Gerufenen durch eine Reihe von Sprachmitteln variiert werden.

Um einen Prozeß in die Lage zu versetzen, gleichzeitig an mehreren seiner Nachrichteneingänge Aufrufe zu erwarten, gibt es die Auswahlanweisung (5–5).

```
select
  [when BEDINGUNG = >]
    AUSW_ALTERNATIVE
  [or [when BEDINGUNG = >]
    AUSW_ALTERNATIVE]*
  [else ANWEISUNGSFOLGE]             (5–5)
end select;

AUSW_ALTERNATIVE ::=
  ANNAHME_ANW [ANWEISUNGSFOLGE]
```

Die Auswahlanweisung besteht aus Alternativen, die mit Bedingungen versehen sein können ("guarded commands" [26]). Bei der Ausführung der Auswahlanweisung

werden zuerst in beliebiger Reihenfolge die Bedingungen der Alternativen berechnet. Die Alternativen, deren Bedingungen wahr sind, werden als "offen" bezeichnet. Eine Alternative, die nicht mit einer Bedingung versehen ist, ist stets offen. Wenn in den offenen Alternativen ein oder mehrere Gespräche möglich sind, dann wird ein beliebiges davon ausgeführt. Folgt innerhalb der so ausgewählten Alternative noch eine ANWEISUNGSFOLGE, dann wird diese als nächstes ausgeführt, und damit ist dann die Ausführung der Auswahlanweisung beendet. Ist mindestens eine Alternative offen und kein Gespräch in den offenen Alternativen möglich und kein else-Teil vorhanden, dann wartet der betreffende Prozeß, bis ein Gespräch in einer der offenen Alternativen möglich ist. Ist ein else-Teil vorhanden und keine Alternative offen oder in den offenen Alternativen kein Gespräch möglich, dann wird der else-Teil ausgeführt. Sind alle Alternativen geschlossen und kein else-Teil vorhanden, dann wird die Ausnahmebedingung SELECT_ERROR gesetzt.

Auf der Seite des Rufers gibt es ebenfalls die Möglichkeit, anstelle des Wartens auf ein Gespräch einen else-Teil auszuführen. Der entsprechende Aufruf ist in (5–6) dargestellt.

```
select
  AUFRUF_EINES_NACHRICHTENEINGANGS
  [ANWEISUNGSFOLGE]
else                                        (5–6)
  ANWEISUNGSFOLGE
end select;
```

Außerdem kann sowohl auf der Seite des Rufers als auch des Gerufenen die Zeitspanne, während der auf ein Gespräch gewartet wird, begrenzt werden (time out).

5.2 Direkte Prozeßsteuerung

Bei der direkten Steuerung der Prozesse kommen die Operationen Erzeugen, Starten, Verzögern, Beenden und Abbrechen vor. Prozesse können in einem Ada-Programm sowohl direkt vereinbart als auch durch die Ausführung von new geschaffen werden. Direkt vereinbarte Prozesse werden bei Abarbeitung der Vereinbarung erzeugt und am Ende des betreffenden Vereinbarungsteils gestartet. Prozesse, die durch die Ausführung von new geschaffen werden, werden dabei erzeugt und sofort gestartet.

Das Verzögern wird durch eine Anweisung der Form (5–7) bewirkt.

```
delay AUSDRUCK;                             (5–7)
```

Der Wert von AUSDRUCK gibt die Verzögerungszeit in Sekunden an.

Ein Prozeß wird beendet, wenn er das Ende seines Rumpfes erreicht und alle von ihm abhängigen Prozesse beendet sind. Eine zweite Möglichkeit des Beendens besteht in der Alternative "terminate;" innerhalb einer Auswahlanweisung (5–5).

Durch eine Anweisung der Form (5–8) wird der betreffende Prozeß abgebrochen.

```
abort PROZESS_NAME;                         (5–8)
```

Infolge der Blockstruktur von Ada-Programmen ist das Verhalten am Blockende [23:224] zu regeln. In Ada kann ein Prozeß einen Block nur dann verlassen, wenn alle davon abhängigen Prozesse beendet sind.

6 Programmschemata

Bei den Programmschemata in Ada (generic units) handelt es sich um parametrisierbare Programmschablonen analog zu den sogenannten Makros in Assemblersprachen [27; 28]. Solche Mechanismen wurden auch früher bereits in höheren Programmiersprachen verwendet [29; 30; 31]. Im Unterschied zu den oben erwähnten Mechanismen können in Ada nur bestimmte Programmeinheiten als Programmschemata formuliert werden, während z. B. in einem Makro in Assemblersprache ein beliebiges Programmstück enthalten sein kann.

Ähnlich wie die Verwendung von Makros führt auch die Verwendung von Programmschemata zu einer Schreib- und Leseersparnis. Außerdem kann sich bei der Verwendung von Programmschemata auch eine Verkürzung des Objektprogrammes ergeben. Wegen der Maschinenunabhängigkeit und Übertragbarkeit von Ada selbst ermöglicht das Konzept der Programmschemata darüber hinaus die Formulierung von parametrisierten und damit anpaßbaren Problemlösungen (insbesondere in der Form von Paketschemata). Damit ist ein Schritt in die Richtung einer "Makroprogrammierung" (programming-in-the-large [16]) getan, d. h. des Zusammensetzens von großen Programmen aus vorgefertigten und bewährten Programmbausteinen. *Ichbiah* sieht dies als den hauptsächlichen Beitrag von Ada zur Verbesserung der Programmerstellung an und erwartet das Entstehen einer Programmbausteine-Industrie, bei welcher Programmbausteine nach Katalog bestellt werden können.

Als Programmschemata können in Ada formuliert werden:

- Unterprogramme
- Pakete

Ein Schema hat allgemein den in (6–1) angegebenen Aufbau.

```
generic
  SCHEMA_PARAMETER                          (6–1)
SPEZIFIKATION
```

Als SPEZIFIKATION kann eine Unterprogramm- oder eine Paketspezifikation auftreten.

Es gibt vier Sorten von Schemaparametern:

- Konstante
- Variable
- Typ
- Unterprogramm

In (6–2) ist ein Prozedurschema angegeben (bestehend

aus Spezifikation und Rumpf), aus welchem man Prozeduren zum Vertauschen der Werte zweier Variablen ableiten kann.

```
generic
  type Elem_Typ is private;
procedure VERTAUSCHEN(A,B: in out Elem_Typ);
                                                    (6–2)
procedure VERTAUSCHEN(A,B: in out Elem_Typ) is
  Z:Elem_Typ;
begin
  Z:=A; A:=B; B:=Z;
end VERTAUSCHEN;
```

Konkrete Vertauschprozeduren kann man aus dem Schema durch Exemplarbildung (generic instantiation) ableiten. Dies ist in (6–3) für zwei verschiedene Elementtypen gezeigt.

```
procedure TAUSCHEN is
  new VERTAUSCHEN (Elem_Typ => INTEGER);
                                                    (6–3)
procedure TAUSCHEN is
  new VERTAUSCHEN (CHARACTER);
```

Die in (6–3) definierten Prozeduren können dann, wie in (6–4) gezeigt, verwendet werden.

```
TAUSCHEN (I_Var1, I_Var2);
                                                    (6–4)
TAUSCHEN (Char_Var2, Char_Var1);
```

7 Programmierumgebung

Bei der ingenieurmäßigen Erstellung von Programmen werden außer dem Sprachübersetzer eine Vielzahl anderer Werkzeuge eingesetzt. Letztlich können alle Phasen dieses Erstellungsprozesses von der Anforderungsanalyse bis zur Wartung durch rechnergestützte Hilfsmittel unterstützt werden. Die Menge dieser Hilfsmittel kann man als Programmierumgebung (in [32] "Software-Produktions-Umgebung") bezeichnen.

Während die Sprache Ada selbst im Vergleich zu herkömmlichen Sprachen als Konzept aus der Programmierumgebung die getrennte, aber nicht unabhängige Übersetzung von Programmeinheiten enthält (s. Abschnitt 2), wird in einem zweiten Projekt versucht, auch die Programmierumgebung unter der Bezeichnung APSE (Ada Programming and Support Environment) zu vereinheitlichen. Ähnlich wie bei der Entwicklung von Ada [3] werden dafür Anforderungskataloge aufgestellt, von denen bisher zwei erschienen sind: "Pebbleman" [33] und "Stoneman" [34].

Die Situation auf dem Gebiet der Programmierumgebungen unterscheidet sich allerdings von der der Programmiersprachen insofern, als auf dem Gebiet der Programmierumgebungen noch mehr Forschung getrieben werden muß, um ein ähnliches Ergebnis zu erzielen, wie es bei der Definition von Ada selbst erzielt wurde. Die Sprache Ada kann als eine Konsolidierung der Forschungsergebnisse auf dem Gebiet der Programmiersprachen der 70er Jahre angesehen werden. Bei dieser Konsolidierung konnte auf wesentlich mehr Forschungsergebnisse zurückgegriffen werden, als dies auf dem Gebiet der Programmierumgebungen derzeit möglich ist. Für das Gebiet der Programmiersprachen sei erinnert an Pascal [7; 6], LIS [35; 36], MESA [37], ALPHARD [38; 39], EUCLID [52], CLU [40], BALG [41], Concurrent Pascal [42; 43], Modula, Modula-2 [44; 45] und Gypsy [46; 47].

Im Stoneman-Vorchlag [35] wird eine dreistufige Gliederung der Programmierumgebung vorgeschlagen:

KAPSE (=Kernel APSE) enthält die Hilfsmittel und Funktionen, die zum Ausführen und Testen von Ada-(Objekt-)Programmen notwendig sind. Dazu gehören folgende Funktionsbereiche:

a) Ada-Laufzeitsystem (inkl. Ein/Ausgabe)
b) Inspektion der abstrakten Syntax einer Übersetzungseinheit.
c) Inspektion der Zwischensprachendarstellung einer Übersetzungseinheit
d) Inspektion und Manipulation eines laufenden Ada-Programmes
e) Inspektion der Symboltabelle einer Übersetzungseinheit
f) Inspektion und Manipulation der Programmbibliothek im Sinne der Ada-Sprachdefinition [8: 10.4].
g) Steuerung einer dialogfähigen Benutzerstation

Die Bereiche b) bis f) sollen jeweils als Ada-Pakete definiert werden. Wenn die Schnittstellen dieser Pakete einheitlich festgelegt werden könnten, wäre eine weitgehende Übertragbarkeit der darauf aufbauenden höheren Schichten der Programmierumgebung (MAPSE und APSE) gewährleistet, da diese höheren Schichten ausschließlich in Ada programmiert werden sollen. Hier stellt sich dann die Frage, wie der relativ statische Programmbegriff von Ada mit dem Wunsch nach dynamischer Umkonfigurierung einer solchen Programmierumgebung in Einklang gebracht werden kann.

MAPSE (=Minimal APSE) besteht aus KAPSE und einer Reihe darauf aufbauender Werkzeuge:

a) Texteditor
b) Programmformatierer
c) Übersetzer
d) Programmbinder
e) Programmlader
f) Analysator für statische Programmstruktur
g) Hilfsmittel zur Überwachung und Steuerung der Ausführung eines Programmes
h) Hilfsmittel zur Benutzung der KAPSE-Funktionen von einer Benutzerstation aus
i) Dateiverwalter
j) Kommandointerpretierer
k) Konfigurationsverwalter

Diese Werkzeuge erlauben die Erstellung, Ausführung und Verwaltung von Ada-Programmen.

APSE kann außer MAPSE noch folgende Werkzeuge enthalten:

a) Ada-Programmeditor
b) Dokumentationssystem
c) System zur Projektsteuerung
d) System zur Versionskontrolle und -verwaltung
e) System zur Messung von Programmabläufen
f) System zur Fehlerdokumentation
h) System zur Bearbeitung von Anforderungsspezifikationen
i) Entwurfssystem
j) Programmbeweiser
k) Leistungsfähige Übersetzer
l) Leistungsfähige Kommandointerpretierer.

Gerade an dieser Liste erkennt man die Bedeutung der eingangs gemachten Bemerkung, daß auf dem Gebiet der Programmierumgebungen noch mehr Forschung getrieben werden muß, bevor ein ähnlicher Stand erreicht wird, wie er auf dem Gebiet der Sprachen vorliegt.

Außerdem erkennt man an den vorstehenden Listen, daß der Aufwand zur Erstellung eines MAPSE oder APSE um ein Vielfaches höher ist als der für einen Sprachübersetzer.

8 Zusammenfassung

In der vorliegenden Arbeit wurden nur die wesentlichen Punkte dargestellt, in welchen Ada Neues bietet, entweder indem es das betreffende Konzept überhaupt erstmalig enthält oder indem es neue Lösungen für bestimmte Probleme enthält. Ein Blick auf die Sprache als Ganzes führt zu dem Urteil, daß Ada, indem es die Ergebnisse der Forschung der 70er Jahre auf dem Gebiet der Programmiersprachen integriert, die Programmiersprache ist, in welcher die meisten wichtigen Sprachelemente in einer guten und kohärenten Form enthalten sind. Im Vergleich zu den eingangs bereits erwähnten "Universalsprachen" PL/1 und Algol 68 kann man feststellen, daß Ada sorgfältiger definiert wurde als PL/1 und daß es realitätsnäher als Algol 68 ist. Die breite Bewährung in der Praxis steht ihm jedoch noch bevor.

Danksagung

Für eine kritische Durchsicht von früheren Versionen des Aufsatzes und wertvolle Hinweise bin ich H. Hummel, K. März, K. Ripken, M. Sommer, C. Stoffel, R. Tobiasch, P. Wehrum und S. Winkler zu Dank verpflichtet.

Literatur

[1] *Broad, William J.:* Pentagon Orders End to Computer Babel. Science 211 (2. Jan. 1981), S. 31–33.
[2] *Carlson, William E.:* Ada: A Promising Beginning. Computer 14, 6 (1981), S. 13–15.
[3] *Fisher, David A.:* DoD's Common Programming Language Effort. Computer 11, 3 (1978), S. 24–33.
[4] *Glass, Robert L.:* From Pascal to Pebbleman ... and Beyond. Datamation 25, 8 (1979), S. 146–150.
[5] *Goos, Gerhard:* ADA: Zweck, Entwicklung und Zukunft einer Programmiersprache. Angewandte Informatik 24, 2 (1982), S. 80–89.
[6] *Jensen, Kathleen; Wirth, Niklaus:* PASCAL – User Manual and Report. Springer, Berlin usw. 1974.
[7] International Standards Organization (ed.): Third Draft Proposal ISO/DP7185, ISO/TC 97/SC 5 N 678, 1981-11-04.
[8] Department of Defense. Reference Manual for the Ada Programming Language. Washington, D.C., July 1980.
[9] *Barnes, J. G. P.:* An Overview of Ada. Software-Practice and Experience 10 (1980), S. 851–887.
[10] *Barnes, J. G. P.:* Programming in Ada. Addison-Wesley, London etc. 1982.
[11] *Ledgard, Henry:* ADA – An Introduction. Springer, New York etc. 1981.
[12] *Pyle, I. C.:* Die Programmiersprache ADA. Carl Hanser, München usw. 1982.
[13] *Brosgol, Benjamin M.:* Summary of Ada Language Changes. Ada Letters I, 3 (1982), S. 34–43.
[14] *Hoare, C. A. R.:* Communicating Sequential Processes. CACM 21, 8 (1978), S. 666–677.
[15] *Brinch Hansen, Per:* Distributed Processes: A Concurrent Programming Concept. CACM 21, 11 (1978), S. 934–941.
[16] *DeRemer, F. L.; Kron, H. H.:* Programming-in-the-large versus Programming-in-the-small, [48, S. 80–89].
[17] *Goos, Gerhard:* Zum Begriff des Programmoduls. Universität Karlsruhe, Inst. f. Informatik 2, Interner Bericht 1/74.
[18] *Balzert, Helmut:* Vergleichende Betrachtung modularer Sprachkonzepte, [49, S. 45–72].
[19] *Wirth, Niklaus:* The Module: a system structuring facility in high level programming languages, in: Tobias, Jeffrey M. (ed.): Language Design and Programming Methodology, Springer, Berlin usw. 1980.
[20] *Sale, A. H. J.:* Forward-declared Procedures, Parameter-lists and Scope. Software – Practice and Experience 11 (1981), S. 123–130.
[21] *Wichmann, B. A.:* Tutorial Material on the Real Data-Types in Ada. Ada Letters I, 2 (1981/82), S. 15–33.
[22] *Winkler, J. F. H.:* Eine Theorie der Prozeßoperationen. Dissertation, Univ. Karlsruhe, Juli 1979, Hochschul-Verlag, Freiburg 1980.
[23] *Winkler, J. F. H.:* Das Prozeßkonzept in Betriebssystemen und Programmiersprachen I. Informatik Spektrum 2, 4 (1979), S. 219–229.
[24] *Walden, David C.:* A System for Interprocess Communication in a Resource Sharing Computer Network. CACM 15, 4 (1972), S. 221–230.
[25] *Wettstein, H.; Becker-Weimann, K.; Winkler, J. F. H.; Wosnitza, H.:* Ein modernes, modulares Betriebssystem für Prozeßrechner und seine Generierung. Gesellschaft für Kernforschung, Karlsruhe, Bericht PDV–E71, Juni 1976.
[26] *Dijkstra, Edsger W.:* Guarded Commands, Nondeterminism and Formal Derivation of Programs. CACM 18, 8 (1975), S. 453–457.
[27] *Kent, William:* Assembler-Language Macroprogramming: A Tutorial Oriented Toward the IBM/360. Computing Surveys 1, 4 (1969), S. 183–196.
[28] *Barron, D. W.:* Assembler und Lader. C. Hanser, München 1970.
[29] *Cheatham, T. E. Jr.:* The introduction of definitional facilities into higher level programming languages. Fall Joint Computer Conference 1966, S. 623–637.
[30] IBM Corp. (ed.). IBM System/360 Operating System, PL/1 (F) Language. Reference Manual. Order No. GC28-8201-4, 5. ed. Dec. 1972.
[31] *Leroy, H.:* A macro-generator for ALGOL. Spring Joint Computer Conference 1967, S. 663–669.
[32] *Hausen, H. L.; Müllerburg, M.:* Software-Produktions-Umgebungen: Entwicklungsstand und Trends, in: Goos, G. (Hrsg.), Werkzeuge der Programmiertechnik, Springer, Berlin usw. 1981, S. 1–27.
[33] Department of Defense: Requirements for the programming environment for the common high order language. PEBBLEMAN Revised, January 1979.
[34] Department of Defense: Requirements for Ada Programming Support Environments "Stoneman", February 1980.
[35] *Ichbiah, J. D.; Rissen, J. P.; Heliard, J. C.; Cousot, P.:* The System Implementation Language LIS. Reference Manual. Technical Report 4549 E/EN December 1974, CII, Louveciennes.
[36] Siemens AG (Hrsg.): LIS Reference Manual. 1. Edition, München, 1978.
[37] *Mitchell, James, G.; Mayburg, William; Sweet, Richard:* Mesa Language Manual. Xerox, Palo Alto Research Center, Palo Alto 1979, Version 5.0, CSL-79-3.
[38] *Wulf, William A.; London, Ralph L.; Shaw, Mary:* An Introduction to the Construction and Verification of Alphard Programs. IEEE Trans. on Software Engineering 2, 4 (1976), S. 253–265.
[39] *Shaw, Mary; Wulf, William A.; London, Ralph L.:* Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators. CACM 20, 8 (1977), S. 553–564.

[40] *Liskov, Barbara:* An Introduction to CLU. [50, S. 139–156].
[41] *Goos, Gerhard:* Die Programmiersprache BALG – vorläufige Fassung – Universität Karlsruhe, Fakultät für Informatik, Bericht 6/75.
[42] *Brinch Hansen, Per:* The Architecture of Concurrent Programs. Prentice Hall, Englewood Cliffs, 1977.
[43] *Brinch Hansen, Per:* Konstruktion von Mehrprozeßprogrammen. R. Oldenbourg, München, Wien 1981.
[44] *Wirth, N.:* Modula: a Language for Modular Multiprogramming. Software – Practice and Experience 7 (1977), S. 3–35.
[45] *Wirth, Niklaus:* MODULA-2. Eidgenössische Technische Hochschule Zürich, Institut für Informatik, Bericht 36, March 1980.
[46] *Ambler, Allen L.; Good, Donald I.; Browne, James C.; Burger, Wilhelm F.,* et al.: GYPSY: A Language for Specification and Implementation of Verifiable Programs. SIGPLAN Not. 12, 3 (1977) S. 1–10.
[47] *Good, Donald I.; Cohen, Richard M.; Hunter, Lawrence W.:* A Report on the Development of Gypsy, [51, S. 116–122].
[48] *Schneider, H.-J.; Nagl, M.* (Hrsg.): Programmiersprachen – 4. Fachtagung der GI. Springer, Berlin usw. 1976.
[49] *Alber, Klaus* (Hrsg.): Programmiersprachen – 5. Fachtagung der GI. Springer, Berlin usw. 1978.
[50] *Schuman, Stephen A.* (ed.): New directions in algorithmic languages 1975. Institut de recherche d'informatique et d'automatique, Rocquencourt, 1976.
[51] ACM (ed.). Proceedings 1978 Annual Conference. ACM, New York 1978.
[52] *Lampson, B. W.; Horning, J. J.; London, R. L.; Mitchell, J. G.; Popek, G. J.:* Report On The Programming Language Euclid. SIGPLAN Notices 12, 2 (1977).
[53] *Maych, Brian:* Problem Solving with Ada. J. Wiley and Sons, New York etc. 1982.
[54] *Stratford-Collins, M. J.:* Ada – A Programmer's Conversion Course. E. Horwood Publishers, Chichester etc. 1982.

Aus der COMPUTER-PRAXIS

Die Förderung der Informationsverarbeitung durch den Bundesminister für Forschung und Technologie

The promotion of information processing technologies by the Federal Ministry for Research and Technology

Elektron. Rechenanl. 24 (1982), H. 4, S. 186–189
Manuskripteingang: 4. Mai 1982

von G. MARX
Bundesminsterium für Forschung und Technologie, Bonn

Nach einer knappen Schilderung der wirtschaftlichen Bedeutung der Informationsverarbeitung wird ein Überblick über die gegenwärtige Förderung gegeben. Die Schwerpunktthemen sind die Softwaretechnologie, die Systemtechnik, die Informationsverarbeitung für Büro und Verwaltung sowie allgemeine Analysen und Prognosen. In einem Ausblick werden künftige Themen und die Förderinstrumente diskutiert.

A brief description is given of the economic importance of information processing. This is followed by a survey of current promotion activities, the chief topics being software technology, systems engineering, information processing for use in offices and administration, as well as general analyses and forecasts. Subjects for future promotion and appropriate promotion instruments are also discussed

1 Wirtschaftliche Bedeutung der Informationsverarbeitung

Die Informationsverarbeitung war in den letzten 10 Jahren einer der größten Wachstumsmärkte. Während sich das Bruttosozialprodukt der Bundesrepublik Deutschland von 1970 bis 1980 verdoppelte, wuchs der Markt für Informationsverarbeitung um das Fünfzehnfache. Der Wert der im Inland installierten Computer wird 1981/82 die 50-Milliarden-Marke überschreiten. Der Antrieb für diese Entwicklung liegt nicht nur in der Möglichkeit, mit dieser Technik Kosten zu senken und die Produktivität zu steigern, sondern auch in den vielfältigen Chancen, die Qualität von Gütern und Dienstleistungen zu verbessern und neue Produkte und Dienste zu schaffen.

Die Datenverarbeitungsindustrie in der Bundesrepublik Deutschland hat sich nach Angaben des ZVEI von 1975 bis 1980 von 3,5 Milliarden DM auf 6,8 Milliarden DM nahezu verdoppelt. Davon wurden 1980 etwa 60% (4,3 Milliarden DM) exportiert. Gleichzeitig wurden Datenverarbeitungssysteme in Werte von rund 5 Milliarden DM importiert. Die gleiche Quelle weist für die Büromaschinenindustrie in den letzten 5 Jahren nur ein Wachstum von knapp 5% aus. Allerdings umfaßt die Sta-