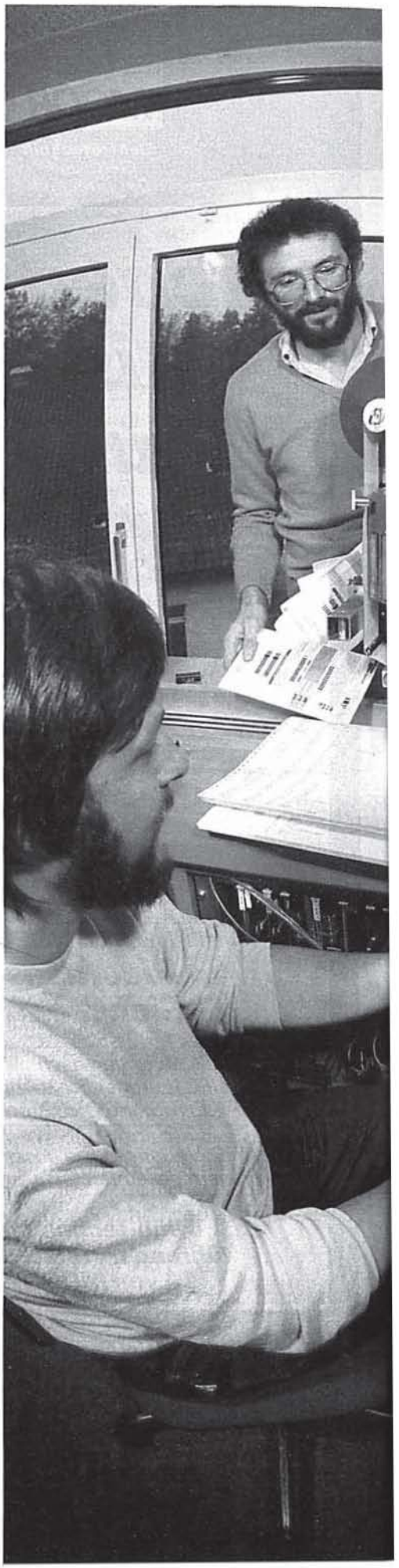
Der folgende Aufsatz behandelt Sprachen mit Modul- und Prozeßkonzept sowie interpretierbare Programmiersprachen und schließt mit zusammenfassenden Argumenten den Beitrag «Sprachen für die Programmierung — eine Übersicht» ab. Der erste Teil der Serie ist in «TR» Nr. 9 vom 25. Februar 1985 erschienen, der zweite in «TR» Nr. 13 vom 26. März 1985.

Modulare und interpretierbare Programmiersprachen

Von Jochen Ludewig



Nach einer Programmänderung muß ein PASCAL-Programm jeweils vollständig neu übersetzt werden. Bei großen Programmen ist dies sehr aufwendig, bei sehr großen Systemen (beispielsweise 100 000 Zeilen) praktisch unmöglich.

Separate Compilierung, MODULA

Abhilfe schafft die Möglichkeit, Teile des Programms *separat* zu übersetzen. Dazu wird es in Moduln (FORTRAN: Hauptprogramm und Subroutines) zerschnitten, die einzeln compiliert werden können (Bild 1).
Dadurch wird der Aufwand bei Programmänderungen erheblich verringert, und die Verwendung vorhandener Programme und Programmbibliotheken wird erleichtert.
Bei der separaten Compilierung tritt das Problem auf, nicht mehr das gesamte Programm auf Konsistenz prüfen zu können. So läßt es sich etwa in FORTRAN nicht automatisch verhindern, daß das folgende, aus zwei Übersetzungseinheiten bestehende Programm ohne Fehlermeldung akzeptiert wird, obwohl Parameter falschen Typs in falscher Zahl übergeben werden.

```
CALL SUB1 (1, 4, 16, 64, 10.24)
END

SUBROUTINE SUB1 (X)
WRITE (6, 1) X
FORMAT (E12.4)
RETURN
END
```

Viele Jahre wurde das Problem der separaten Übersetzung in den Sprachen der ALGOL-Familie ausgeklammert, da man keine Möglichkeit zu einer sauberen Behandlung hatte. Erst mit MODULA (*modu*lar *la*nguage) legte WIRTH 1976 ein brauchbares Konzept vor. MODULA entstand auf der Grundlage von PASCAL zunächst als Spezialsprache zur Systemprogrammierung; daher fehlte eine Reihe von PASCAL-Konzepten. Bald wurde es jedoch weiterentwickelt zu MODULA-2 [17] und dabei wieder angereichert, so daß die neue Fassung wesentlich näher bei PASCAL liegt und im wesentlichen (semantisch) eine Erweiterung darstellt. Alle folgenden Aussagen beziehen sich auf MODULA-2; das ursprüngliche MODULA spielt praktisch keine Rolle mehr.
In MODULA-2 werden Moduln (die Übersetzungseinheiten) jeweils durch zwei Teile beschrieben, einen *Definitionsmodul*, der die äußere Schnittstelle definiert, und einen *Implementierungsmodul*, der das eigentliche Programm enthält. Andere Moduln haben grundsätzlich nur zu den im Definitionsmodul enthaltenen, explizit *exportierten* Informationen Zugang. Bei Übersetzung eines Definitionsmoduls wird ein *Symbolfile* erzeugt, auf das der Compiler später zurückgreift. Dadurch kann die Konsistenz auch über die einzelne Übersetzung hinaus sichergestellt werden. Durch die Trennung zwischen Definition und Implementierung wird auch die gleichzeitige Arbeit an mehreren Moduln erleichtert.

Beispiel: eine Warteschlange als MODULA-Modul*

```
DEFINITION MODULE Buffer;
  EXPORT QUALIFIED put, get, empty, full;

(* Warteschlange mit put zum Zufügen eines Elements, get zum Entfernen
   des ältesten Elements, empty und full geben an, ob die Schlange ganz
   leer bzw. ganz voll ist.                                            *)

  VAR empty, full: BOOLEAN;
  PROCEDURE put (x: CARDINAL);
  PROCEDURE get (VAR x: CARDINAL);
END Buffer.
```

Der Definitionsmodul enthält also die für einen anderen Modul notwendigen syntaktischen Informationen, die Funktionalität ist nur durch einen Kommentar beschrieben. Der zugehörige Imple-

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

* nach [17], S. 82 f.

mentierungsmodul kann beispielsweise so aussehen:

```
IMPLEMENTATION MODULE Buffer;
  CONST N = 100;
  VAR   in, out : [0..N-1];
        n       [0..N];
        buf: ARRAY [0..N-1] OF CARDINAL;

  PROCEDURE put (x: CARDINAL);   (* der Schlange ein Element hinzufuegen *)
  BEGIN
    IF n < N
    THEN
      buf [in] := x;
      in       := (in + 1) MOD N;
      n        := n + 1;
      full     := n = N;
      empty    := FALSE
    END
  END put;

  PROCEDURE get (VAR x: CARDINAL); (* der Schlange ein Element entnehmen *)
  BEGIN
    IF n > 0
    THEN
      x        := buf [out];
      out      := (out + 1) MOD N;
      n        := n - 1;
      empty    := n = 0;
      full     := FALSE
    END
  END put;

BEGIN (* Initialisierung *)
  n := 0; in := 0; out := 0;
  empty := TRUE; full := FALSE
END Buffer.
```

Andere Moduln müssen nun die benötigten Namen explizit importieren, zum Beispiel mit

```
FROM Buffer IMPORT put, get, empty, full;
```

Der Prozeduraufruf «get (17)» in einem anderen Modul erzeugt bei der Übersetzung eine Fehlermeldung, denn der Compiler «weiß» aus dem Symbolfile «Buffer», daß «get» einen VAR-Parameter benötigt, daß also eine Konstante unzulässig ist.

Information Hiding und abstrakte Datentypen

Das Programm oben ist das klassische Beispiel für das Kapseln einer lokalen Information in einem speziellen Modul. Die Konstante N und die Variablen buf, n, in und out sind modullokal, das heißt, sie werden nicht exportiert und sind daher für die anderen Moduln nicht sichtbar. Dadurch wird die *Realisierung* der Warteschlange zu einer privaten Angelegenheit dieses einen Moduls. Sie kann ohne Schwierigkeiten geändert werden (beispielsweise durch Verwendung einer verketteten Liste anstelle eines Feldes); solange die äußere Schnittstelle eingehalten wird, bleibt das Zusammenspiel mit anderen Moduln sicher unverändert.
Die Vorteile dieses Verfahrens, das von PARNAS 1971 unter der Bezeichnung des *information hiding* beschrieben wurde, zeigen sich leicht am folgenden Vergleich:

Bei einer (sehr einfachen) Bank seien die folgenden Operationen möglich: Empfang und Veranlassung bargeldloser Zahlungen, Ein- und Auszahlung von Bargeld und Abfrage des Konto-

Das wichtigste Kennzeichen eines guten Programmierers ist seine Disziplin . . .
(Foto: HR. Bramaz)

standes. Der Kunde kann alle diese Operationen am Bankschalter oder schriftlich auslösen, ohne sich um die interne Organisation zu kümmern. Grundsätzlich ist es für ihn ohne Belang, ob sein Geld in einem speziellen Fach aufbewahrt wird oder ob es mit allen anderen Geldern zusammengelegt ist und nur eine Buchführung die Zuordnung zu einzelnen Konten speichert. Nehmen wir an, der Kunde habe statt dessen Kenntnisse der internen Organisation und die Möglichkeit des direkten Zugriffs, zum Beispiel indem er in Selbstbedienung Geld in eine bestimmte Kasse einlegt oder daraus entnimmt. Zum einen wäre es bei einer Unstimmigkeit der Abrechnung so gut wie unmöglich, den Grund festzustellen. Aber selbst mit den ehrlichen und fehlerfrei arbeitenden Kunden könnte das Verfahren auf Dauer nicht funktionieren, weil die Bank bei einer Umstellung ihrer internen Organisation einer unübersehbaren Zahl von Kunden die Änderung mitteilen und ihre Beachtung überwachen müßte.

Die gleichen Probleme gibt es auch in Programmen: Wo Daten in einem großen Programmsystem allgemein zugänglich sind, kann nicht mehr ohne weiteres festgestellt werden, ob sie nur korrekt gebraucht und verändert werden. Wird die Darstellung der Daten verändert, so sind die Folgen nicht zu überblicken. Die Kapselung von Daten, wie sie oben durch die Operationen get und put erreicht ist, wird daher heute allgemein als eines der wichtigsten Programmierprinzipien betrachtet. Es bewährt sich auch dort ausgezeichnet, wo Peripheriegeräte angesprochen werden; alle geräteabhängigen Merkmale wie die Zahlendarstellung werden in einen speziellen Modul gelegt, und nur dieser muß geändert werden, wenn das Gerät durch ein anderes ersetzt wird.

Aus dem Vergleich mit der Bank läßt sich auch ein anderes Merkmal dieses Vorgehens ablesen; welches vor allem bei großen Systemen wichtig ist. Eine Haushaltskasse kann auch nach dem Selbstbedienungsverfahren verwaltet werden, so wie ein überschaubares Programm auch ohne «information hiding» noch handhabbar ist. Daher können (unvermeidlich kleine) Beispiele die Vorteile nur andeuten.

In der Sprache SIMULA (1967) waren in Form der CLASS zum erstenmal Konstrukte vorgesehen, mit deren Hilfe Datentypen und darauf zulässige Operationen gekapselt werden konnten. Daher spricht man auch von *Abstract Data Types;* die spezielle Realisierung der Daten wird durch die Abstraktion unsichtbar, nur ihre funktionale Schnittstelle ist durch die Operationen zugänglich.

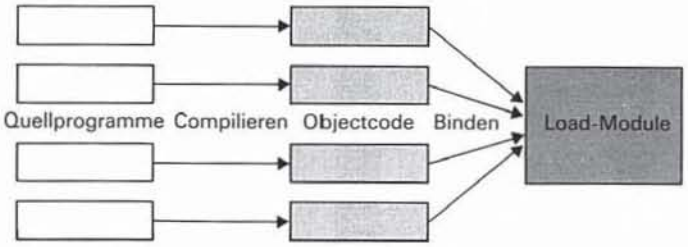


Bild 1. Die Moduln eines Programms lassen sich einzeln compilieren.

Das Beispiel der Bank zeigt, warum man von *Typen* spricht: Die Operationen wie Einzahlung usw. gelten nicht für ein spezielles Konto, sondern für *alle* Objekte des Typs Konto. In MODULA und ADA wurde dieses Prinzip aus SIMULA weiterentwickelt.

Parallele Prozesse

Auf einem einfachen Rechner des VON-NEUMANN-Typs läuft jeweils ein einziges Programm bis zum Ende, dann folgt das nächste. Dieser streng sequentielle Betrieb hat folgende Nachteile:

- Bei der Ein- und Ausgabe muß die Zentraleinheit (CPU) relativ lange warten, oft den größten Teil der Zeit, da die Peripheriegeräte Zeiten in der Größenordnung von Hundertstel- bis Zehntelsekunden benötigen, während die CPU weniger als eine Mikrosekunde pro Operation braucht.
- Wenn mehrere technische Abläufe überwacht werden sollen und die Antwortzeit des Rechners kritisch ist, sind vielfach Reaktionen außerhalb des sequentiellen Programmablaufs nötig.
- Wo an einem System mehrere Benutzer arbeiten, ist die Beschreibung durch einen einzigen bedienenden Prozeß im Rechner äußerst kompliziert.
- Rechner mit mehreren Prozessoren, die parallel arbeiten können, lassen sich sequentiell überhaupt nicht ausnutzen.

Aus diesem Grunde wurde das Konzept der *parallelen Prozesse* geschaffen. Diese arbeiten auf verschiedenen Prozessoren (echt parallel) oder auf einem einzigen in nicht vorher definierter zeitlicher Verzahnung (quasiparallel). Für die Programmierung ist dieser Unterschied unwesentlich; wir gehen daher nachfolgend vom allgemeineren Fall, der echten Parallelität, aus.

Solange die parallelen Prozesse völlig unabhängig voneinander sind, entstehen keine Probleme, denn die Prozessoren ignorieren einander. Erst durch die Notwendigkeit, die Prozesse miteinander kommunizieren zu lassen (zum Zweck des Datenaustauschs oder zur Koordination der Zugriffe auf gemeinsame Betriebsmittel), wird die Sache kompliziert.

Untersuchungen in den sechziger Jahren haben zwei wesentliche Grundsätze für die Organisation paralleler Prozesse ergeben:

- Es darf *unter keinen Umständen* zu einer Situation kommen, in der zwei oder mehr Prozesse wechselseitig aufeinander warten («Verklemmung», «Deadlock», «deadly embrace»).
- Es ist unzulässig, Annahmen darüber zu machen, wie schnell parallele Prozesse fortschreiten, das heißt, die Geschwindigkeit darf keine Rolle spielen, weder wenn ein Prozeß extrem schnell oder langsam ist (führt sonst zu den «racing conditions») noch wenn alle exakt gleich schnell sind (verursacht eventuell das ständige höfliche «after-you»-Warten).

Die Programmabschnitte, in denen parallele Prozesse auf gemeinsame Variablen oder Betriebsmittel zugreifen, werden als *kritische Abschnitte* («critical regions») bezeichnet. DIJKSTRA hat 1968 ein Verfahren angegeben, diese durch sogenannte *Semaphore* zu schützen. Vorbild (und Stifter des Namens) waren die Signale der Eisenbahntechnik.

Ein Semaphor S ist eine spezielle Variable mit ganzzahligem, niemals negativem Wert. Auf sie sind nur zwei Operationen zugelassen, wait (S) und send (S) (bei DIJKSTRA P(S) und V(S)). Send (S) bedeutet, daß der Wert von S um 1 erhöht wird. Bei wait (S) wird er um 1 gesenkt, falls er größer als Null war; andernfalls wird die Wait-Operation verzögert, bis S durch Send-Operationen anderer Prozesse positiv geworden ist.

Beispiel: Erzeuger und Verbraucher (in einer MODULA-ähnlichen Notation)

```
SEMAPHOR S := 0;

PROCESS hersteller;                 | PROCESS verbraucher;
LOOP                                | LOOP
  (* Ware herstellen, hier         |   wait (S);      (* wartet auf Ware *)
     nicht ausgefuehrt *)           |   (* Ware verbrauchen, hier
  send (S) (* liefert Ware *)       |      nicht ausgefuehrt *)
END LOOP;                           | END LOOP;
```

Beide Prozesse können *asynchron* arbeiten, jedoch wird der Verbraucher verzögert, wenn er den Hersteller überholen will.
Im Sinne des Maschinencodes bestehen wait und send im allgemeinen aus mehreren Einzeloperationen. Werden mehrere solche Zugriffe auf denselben Semaphor parallel ausgeführt, so kann das Endergebnis verfälscht werden, wie das folgende Beispiel (links send (S), rechts wait (S), Zeitachse in der Mitte) verdeutlicht.

Das Ergebnis ist falsch, denn es spiegelt nur die Wait-Operation. Es ist daher notwendig, die Operationen send und wait *unteilbar* zu machen. Dies gilt auch für jeden anderen Mechanismus zur Prozeßkommunikation: Immer ist eine unteilbare Operation für die Realisierung erforderlich. Bei quasiparallelen Systemen kann dies auch durch das Betriebssystem garantiert werden, bei echter Parallelität ist eine geeignete Maschinenoperation notwendig.
In den letzten Jahren wurden verschiedene komfortablere und/oder sicherere Mechanismen entwickelt, darunter die *Path-Expressions* von CAMPBELL und HABERMANN, die *Monitore* von HOARE und das *Rendezvous* in ADA.

Die Programmiersprache ADA

Das US-Department of Defense (DoD) wendet für Software jährlich eine Summe von etwa 3 Mia $ auf. Als eine Möglichkeit, diese Kosten zu senken, wurde die Vereinheitlichung der zahlreichen eingesetzten Programmiersprachen (es sollen mit Dialekten etwa 400 sein) identifiziert. So entstand die Programmiersprache ADA, benannt nach der Gräfin AUGUSTA ADA LOVELACE, Tochter LORD BYRONS, die als erste Programmiererin (der nicht realisierten «analytical engine» von BABBAGE) zur Geschichte der Programmierung beigetragen hat.
ADA wurde in einem mehrstufigen Auswahlverfahren von verschiedenen Stellen entwickelt (die Gesamtkosten liegen bisher über 20 Mio $); das Rennen machten schließlich JEAN D. ICHBIAH und seine Mitarbeiter bei Cii Honeywell Bull in Frankreich. Dies erklärt, wieso eine Sprache des DoD so enge Verwandtschaft mit den europäischen Sprachen wie PASCAL und MODULA aufweist.
ADA wurde entsprechend den Bedürfnissen der Auftraggeber für die Programmierung komplexer «embedded systems» entworfen. Dieses Wort bezeichnet Programme, die Teile größerer Systeme, zum Beispiel eines Waffensystems, werden. Natürlich sind die Erfordernisse bei Systemsoftware (beispielsweise Betriebssystemen) und bei industriellen Echtzeitsystemen ganz ähnlich, so daß die Sprache für ein weites Spektrum von Anwendungen in Frage kommt.
ADA soll als Standardsprache praktisch alle anderen Sprachen und Dialekte verdrängen und selbst keine Dialekte bilden. Für das erste ist es notwendig, daß ADA sehr mächtig ist und nicht für manche Probleme ungeeignet. Daher ist ADA eine recht umfangreiche Sprache, die – in vollem Umfang – sicher nur mit erheblichem Aufwand erlernt werden kann. Aber man sollte sich darüber im klaren sein, daß man diesen vollen Umfang natürlich oft nicht benötigt. Wer etwa nur sequentielle Programme schreibt, hat keinen Grund, die Mechanismen zur Prozeßkommunikation zu erlernen. Es wird also erforderlich sein, für Zwecke der Ausbildung Teilmengen der Sprache zu definieren. Diese werden dann aber nicht schwieriger sein als andere Sprachen derselben Mächtigkeit.
Um die Bildung von Dialekten zu verhindern, hat das DoD von Beginn an sehr rigorose Maßnahmen getroffen: ADA ist ein eingetragenes Warenzeichen des DoD und darf daher nur mit dessen Genehmigung verwendet werden. Dadurch behält das DoD die Kontrolle und kann jede Abweichung von der präzisen Definition, die vor kurzem in revidierter Fassung erschienen ist (DoD, 1983), mit dem Entzug des Namens bestrafen. Wer einen ADA-Übersetzer auf den Markt bringen will, muß diesen zunächst dem DoD zur Genehmigung vorlegen. In einer ausgefeilten Prozedur wird er geprüft und freigegeben.
Für den Anwender hat diese Strenge große Vorteile: Zum erstenmal ist sichergestellt, daß Programme wirklich portabel sind, es sei denn, sie verwenden spezielle maschinenabhängige Befehle, die bei der Erstellung von Systemsoftware nicht ganz entbehrlich sind, aber wenigstens sauber gekennzeichnet werden. Auch muß der Programmierer nicht für jede Maschine einen neuen Dialekt lernen.

ADA ist der Sprache MODULA recht ähnlich, den Moduln entspricht das *Package*, bestehend aus *package-specification* (Definitionsteil) und *package-body* (Implementierungsteil). Wie MODULA gestattet auch ADA, maschinennah zu programmieren; kann man sich bereits auf ein Betriebssystem stützen, so bietet ADA mehr höhere Konstrukte an, zum Beispiel für die Synchronisation («Rendezvous») oder für die Behandlung von Ausnahmesituationen (Exception-Handling). Das sogenannte Typing, also die Zuordnung von Typen zu Variablen und anderen Objekten, wurde in ADA noch verfeinert. Auch wenn MODULA zu den sauber definierten Programmiersprachen zählt, ist ADA in diesem Punkt sicher überlegen, vor allem auch durch die Standardisierung der Abnahmeprozedur. Mit den sogenannten *Generics*, vor allem den Generic Packages, bietet ADA ein neues Konzept, das gewisse Möglichkeiten von Makroprozessoren mit dem Komfort und der Sicherheit hoher Programmiersprachen kombiniert.
Pragmatisch betrachtet, wird sich ADA aber nicht wegen seiner (sicher vorhandenen) positiven Merkmale durchsetzen, sondern aufgrund des finanziellen Drucks, den das DoD auf den Markt ausübt. Gegenwärtig sind praktisch alle Rechnerhersteller und viele Softwarehäuser mit der Implementierung von Compilern beschäftigt, 1983 wurde (nach einem experimentellen Übersetzer der New York University) der erste echte ADA-Compiler vom DoD validiert.
Ein wesentlicher Vorzug von ADA, auf den hier nicht näher eingegangen werden kann, besteht im sogenannten «ADA Programming Support Environment» (APSE), durch das erstmals (ähnlich wie im sehr erfolgreichen UNIX-System [18]) auch die Bedienschnittstelle festgelegt ist. Damit sind nicht nur die Programme, sondern auch die Programmierer und die Bediener des Systems «portabel» geworden.

Abschließend soll hier das schon früher benutzte Beispiel der Fakultätsberechnung mit ADA gezeigt werden. Es zeigt sehr charakteristisch, wie das Programm nun auch Problemfälle wie die Eingabe einer negativen oder einer zu großen Zahl sauber behandelt (siehe Exception-Statements am Ende). Daß es dadurch länger wird, ist nicht vermeidbar.

```
with TEXT_IO; use TEXT_IO;      -- Verwendung der Standard-Ein-/Ausgabe
package INT_IO is new INTEGER_IO (integer);
with INT_IO, TEXT_IO, use INT_IO, TEXT_IO;
procedure FAKULTÄT is
begin
  loop
    declare
      ZAHL: integer;
      UNZULAESSIGER_WERT: exception;

      function FAKFUNC (ARGUMENT: integer) return integer is
      begin
        if ARGUMENT >= 1   then return 1;
        else  return ARGUMENT * FAKFUNC (ARGUMENT - 1);
        end if;
      end FAKFUNC;  -- das war das rigorose Herz des Programms

    begin       -- Rahmenprogramm, dient vor allem der Ein- und Ausgabe
      PUT_LINE ("Gib Zahl ein: ");
      GET  (ZAHL);
      if ZAHL < 0 then exit; end if; -- Ausspung aus der Schleife
      if ZAHL > 0 then raise UNZULAESSIGER_WERT; end if;

      PUT ("Fakultaet von "); PUT (ZAHL, WIDTH => 2); PUT (" ist ");
      PUT (FAKFUNC (ZAHL), WIDTH => 6); PUT_LINE (".");

    exception           -- Behandlung von Ausnahmesituationen
      when NUMERIC_ERROR     => PUT_LINE ("zu gross");
      -- bei Zahlenueberlauf entsteht z.B.:
      -- "Fakultaet von 99999 ist zu gross"
      when UNZULAESSIGER_WERT => PUT_LINE ("Zahl ist unzulaessig");
    end;
  end loop;
end FAKULTAET;
```

Zu den Kommentaren, die durch doppeltes Minus eingeleitet werden, folgender Hinweis: Die Kommentare werden in ADA nicht wie in PASCAL und ähnlichen Sprachen durch zwei Symbole umschlossen, zum Beispiel «{. . .}», sondern enden automatisch am Zeilenende wie bei FORTRAN oder COBOL, können allerdings auch in der Zeile beginnen. Programme werden dadurch besser lesbar, und Fehler durch fehlendes Kommentarende-Symbol sind ausgeschlossen.

Die Literatur über ADA ist schon heute sehr umfangreich. [19] bietet einen leichten Zugang, [20, 21] sind sehr zu empfehlen.

Compiler und Interpreter

Das Prinzip der Programmübersetzung ist im ersten Teilbeitrag der Serie in der «TR» Nr. 9 (26. 2. 1985) erklärt worden. Der *Compiler* erhält ein vollständiges Quellprogramm und liefert ein entsprechendes Maschinenprogramm (das im allgemeinen noch gebunden werden muß). Bei der Programm*ausführung* ist der Compiler unbeteiligt.

Interpreter arbeiten anders: Das Programm wird nicht übersetzt, sondern in Quellsprache ausgeführt. Der Interpreter analysiert jeweils den anstehenden Befehl, *interpretiert* ihn und führt ihn aus. Bild 2 zeigt die beiden Verfahren schematisch.

Die Compilierung wird darin durch den oberen, die Interpretation durch den unteren Weg vom Quellprogramm zu den Ergebnissen repräsentiert. Tabelle 1 nennt die wesentlichen Konsequenzen aus der Entscheidung für das eine oder das andere Verfahren.

Die Ausführungszeit ist natürlich bei Interpretation erheblich länger, denn es ist ja bei jedem Schritt eine Übersetzung nötig. Dies ist vor allem dann sehr ineffizient, wenn einige wenige Befehle sehr oft durchlaufen werden. Der Implementierungsaufwand ist bei einem Compiler beträchtlich höher, allerdings vor allem durch einen gewissen Grundaufwand. Der Sprachumfang beeinflußt einen Interpreter weitaus stärker als einen Compiler. Interpreter brauchen weniger Speicherplatz als Compiler, schon deshalb, weil sie keinen Maschinencode erzeugen. Interpretation setzt voraus, daß jede Zeile einzeln (also ohne Kenntnis des Kontexts) interpretierbar ist. Damit sind viele Merkmale höherer Sprachen, wie Deklarationen, Datenstrukturen, Unterprogramme usw., nicht mehr beherrschbar. Das heißt, Interpreter sind für solche Sprachen prinzipiell ungeeignet. In der Praxis werden daher oft Merkmale der Compiler in die Interpreter übernommen.

Zusammenfassend läßt sich sagen: Bei Interpretern steigen der Aufwand zur Implementierung sowie der Zeit- und Speicherplatzbedarf im wesentlichen proportional mit dem Umfang der Sprache, wobei von einer gewissen Komplexität an das interpretierende Verfahren nicht mehr ausreicht. Für Compiler entsteht bei den drei genannten Kenngrößen ein erheblicher Grundaufwand (auch für sehr einfache Sprachen), der dann aber nicht mehr dramatisch ansteigt, wenn die Sprache umfangreicher und komplizierter wird.

BASIC

Die genannten Eigenschaften eines Interpreters deuten darauf hin, daß dieser Ansatz dann vorteilhaft ist, wenn

- die Programmiersprache einfach ist (nicht im Sinne des Gebrauchs, sondern im Sinne ihrer Konzeption)
- der Speicherplatz sehr knapp ist
- die Ausführungszeit der Programme keine wesentliche Rolle spielt

Tabelle 1. Merkmale von Compiler und Interpreter.

	Compiler	Interpreter
Übersetzungszeit	je nach Programmlänge und Compiler verschieden lang	null
Ausführungszeit	kurz	lang
Implementierungsaufwand	hoch	gering
Speicherbedarf	höher	geringer
Übersetzung komplizierterer Strukturen	geeignet	ungeeignet

- der Benutzer interaktiv programmiert, testet und korrigiert und ungern auf die Übersetzung von Programmen wartet

und nicht zuletzt

- der Hersteller des Übersetzers keinen großen Aufwand treiben kann

Damit ist der Bereich der *Kleinrechner* (Personal- oder Home-Computer) charakterisiert. So hat sich vor allem auf diesem Gebiet eine Sprache für die interpretative Verarbeitung, die Sprache BASIC (Beginner's All Purpose Symbolic Instruction Code), als erfolgreichste Programmiersprache erwiesen. BASIC wurde 1965 am Dartmouth College, New Hampshire, von KEMENY und KURTZ entwickelt. (Die nachfolgende Besprechung bezieht sich auf das ursprüngliche BASIC. Aus dieser Sprache hat sich eine solche Vielfalt von Dialekten gebildet, daß hier gar nicht erst versucht werden soll, irgendeinen Überblick zu gewinnen.)

BASIC entstand zu einer Zeit, da es für die interaktive Arbeit keine Bildschirmterminals, sondern nur Fernschreiber (Konsol-Schreibmaschinen) gab. Auf diesen Geräten kann nur zeilenweise editiert werden. Daher beginnt in BASIC jedes Statement mit einer Zeilennummer. Durch diese ist sein Platz im Programm bestimmt (das heißt, die Reihenfolge der Eingabe ist unwesentlich, die Ordnung erfolgt aufgrund der Zeilennummern). Dann folgt eine der folgenden Anweisungen:

```
LET variable = ausdruck                                  | Wertzuweisung |
GOTO zeilennummer                                        | Sprung |
GOSUB zeilennummer                                       | Unterprogr.-Sprung |
RETURN                                                   | Unterpr.-Rücksprung |
IF ausdruck vergleichssymbol ausdruck THEN zeilennummer  | Bedingter Sprung |
FOR variable = ausdruck TO ausdruck STEP ausdruck        | Laufschleife |
NEXT variable                                            | Ende Laufschleife |
READ variable, variable, ....                            | Eingabe, siehe DATA |
PRINT zeichenreihe-oder-ausdruck, zeichenreihe-oder-ausdruck, ....  | Ausgabe |
STOP                                                     | Ende Programm-Lauf |
END                                                      | Ende des Programms |
DIM variable ( integer )                                 | eindim. Feld |
DIM variable ( integer, integer )                        | zweidim. Feld |
DATA zahl, zahl, ....                                    | Konstanten, s. READ |
REM kommentar                                            | Kommentarzeile |
DEF FN zeichen ( variable ) = ausdruck                   | Funktion |
```

Außerdem gibt es elf Anweisungen für Matrixoperationen und einige mathematische Funktionen wie Quadratwurzel, Zufallszahl usw. Eine frühe Erweiterung betraf die Verarbeitung von Strings (Zeichenreihen).

Namen bestehen in BASIC aus nur einem einzigen Buchstaben, auf den noch eine Ziffer folgen darf. Zahlen werden in jedem Fall als Realzahlen behandelt, es gibt also keinen Typ für ganze Zahlen (vergleiche die Ergebnisse im Beispiel unten).

Ein BASIC-Programm zur Berechnung der Fakultätsfunktion sieht so aus:

```
10  DATA 5, 10, 25, 0
20  READ Z
25  IF Z = 0 THEN 80
30  LET A = 1
40  FOR B = 1 TO Z STEP 1
50  LET A = A * B
60  NEXT B
70  PRINT "FAKULTAET VON ", Z, " IST ", A
75  GOTO 20
80  END
```

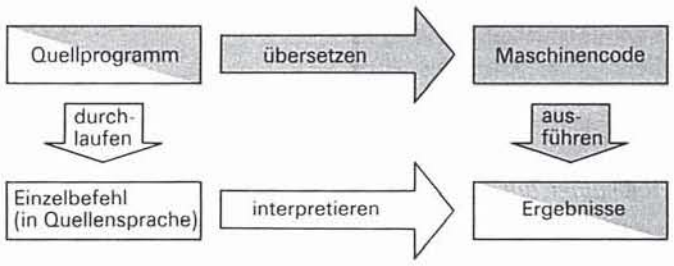


Bild 2. Schematische Darstellung der Verfahrensweise von Compiler und Interpreter.

An den Zeilennummern erkennt man, daß zwei Zeilen nachträglich eingefügt wurden, die Zeilen 25 und 75. Dadurch wurde eine Schleife über mehrere Werte für Z realisiert. Eingelesen wird aus den Daten im DATA-Statement. Ein Lauf des Programms liefert folgende Ergebnisse:

```
FAKULTAET VON          5          IST          120
FAKULTAET VON          10         IST          .3628BE+07
FAKULTAET VON          25         IST          .155112E+26
```

Ein interaktives System läßt sich auf totem Papier schlecht beschreiben, man muß sich vorstellen, daß alle Tätigkeiten wie Eingeben und Editieren, Ausgeben auf dem Terminal und Testen ohne Wechsel der Betriebsart durcheinander ausgeführt werden können. So wird das Programm zeilenweise eingegeben, mit LIST kontrolliert und schließlich mit RUN gestartet. Anschließend kann das Programm durch Eingabe, Ersetzung oder Löschen einzelner Zeilen geändert und erneut gestartet werden. Wegen der ineffizienten Programmausführung gibt es heute vielfach zusätzlich BASIC-Compiler, mit denen ein getestetes Programm in effizienten Code übersetzt werden kann.

Kritik an BASIC

BASIC ist strukturfeindlich. Die Sprache besitzt (immer in ihrer ursprünglichen Form) weder ein echtes Unterprogrammkonzept noch Datentypen (abgesehen von Feldern), es fehlen aussagekräftige Variablennamen und Konstrukte zur syntaktischen Gliederung der Programme.

Die charakteristischen Eigenschaften der Sprache BASIC und des zugehörigen interaktiven BASIC-Systems fördern eine bestimmte Arbeitsweise: Der Programmierer probiert schnell mal etwas aus, flickt ein paar Befehle ein, probiert wieder, ändert hier und dort etwas, bis das Programm zu tun scheint, was er erwartet. Dieses Vorgehen, von Kritikern mit einem abgewandelten amerikanischen Werbespruch als «Code now, think later» beschrieben, steht im scharfen Konflikt mit den Erkenntnissen auf dem Gebiet des Software-Engineering (die Disziplin, die sich damit befaßt, wie man *korrekte* Programme mit möglichst *geringem Aufwand* produziert *und* erhält). Wir wissen heute, daß mehr als zwei Drittel des Personalaufwands der «Wartung» dienen, also der Korrektur und Änderung fertiger Programme. Der Wartungsaufwand wird vor allem durch drei Faktoren beeinflußt:

- durch die Fehlerdichte
- durch die Verständlichkeit des Programms und
- durch die Qualität seiner Strukturierung. Dieser Punkt entscheidet darüber, wie groß der Teil des Programms ist, in dem ein Fehler gesucht oder eine Änderung durchgeführt werden muß

Die interaktive Programmierung erscheint uns gefühlsmäßig so attraktiv, weil wir unmittelbar ein Ergebnis sehen. Wir nehmen dafür in Kauf, daß wir mit einem schlecht (oder eigentlich gar nicht) strukturierten Programm arbeiten. Da komplexe Datentypen und andere Konstrukte, die die Übersicht erhöhen würden, fehlen, können wir unsere Programme kaum weitergeben oder fremde verstehen (eigene Programme sind auch fremde, wenn sie älter sind). Die Fehler hoffen wir durch Probieren zu entdecken, obwohl wir wissen, daß man durch Testen keine sicheren Aufschlüsse erhält.

Aus diesen Erwägungen folgt, daß BASIC eine schlechte Programmiersprache ist. Das gilt auch und vor allem für die Ausbildung. Untersuchungen haben gezeigt, daß die Programmiersprache, die wir als erste erlernt haben, unsere Art zu programmieren in derselben Weise prägt, wie es auf anderen Gebieten unsere Muttersprache tut. Daher ist BASIC der denkbar schlechteste Einstieg in die Programmierung (DAVID GRIES, 1976: «Today thousands of teenagers in the US are having their first algorithmic thoughts polluted by this language.» Der ganze Artikel ist sehr lesenswert [22]).

> *WARNING: The Programmer General has determined that BASIC hacking may be hazardous to the programming health of youngsters. For details see R. L. WEXELBLAT's «The consequence of one's first programming language», Software – Practice and Experience 11 (1981), pp. 733–740.*

Aus «Communications of the ACM», Vol. 27, 1, p. 13.

Zu BASIC gibt es eine Reihe von Lehrbüchern, zum Beispiel von SCHÄRF [23]. Wegen der großen Unterschiede zwischen den BASIC-Dialekten ist es allerdings vorteilhaft, die speziellen Handbücher der jeweiligen Implementierung zu benutzen.

Neben BASIC gibt es noch andere, jedoch weniger weit verbreitete Sprachen für die interaktive Programmierung. Hier ist vor allem APL zu nennen. APL hat wegen seiner äußerst knappen, dem Laien völlig unverständlichen Notation, die einen besonderen Zeichensatz (viele griechische Buchstaben und Sonderzeichen) und damit spezielle Terminals erfordert, eine kleine Gemeinde überzeugter Fans, die sich oft einen Sport daraus machen, komplizierte Probleme mit einem Programm zu lösen, das in eine einzige Zeile paßt (die berüchtigten

«Oneliner»). APL gilt daher unter den «Ungläubigen» als praktisch unlesbar.

Programmqualität, Effizienz und Optimierung

Bei der Kritik an BASIC wurden bereits einige Programmqualitäten genannt. Will man die angestrebten Eigenschaften klassifizieren, so unterscheidet man zwischen denjenigen, die die Arbeit *mit* dem Programm betreffen (beispielsweise Korrektheit, Robustheit, Effizienz), und denen, die charakterisieren, wie schwierig Arbeiten *am* Programm sind (etwa Portabilität, Strukturierung und Verständlichkeit, Qualität der Programmdokumentation). Zwischen den verschiedenen Qualitäten besteht im allgemeinen ein Konflikt, so daß der Programmierer einen Kompromiß suchen muß, mit welchem sich insgesamt das beste Resultat erzielen läßt.

Viele von uns folgen dabei einem starken Bestreben, den Aspekt der Effizienz übermäßig hoch zu bewerten und dadurch vor allem die Qualitäten der zweiten Gruppe zu vernachlässigen. Dies gilt sowohl für die Effizienz der Programmerstellung («schnell zum Laufen bringen») als auch und vor allem für die der Programmausführung. Dies hat hauptsächlich drei Gründe:

- Wir lernen an kleinen Beispielen programmieren, bei denen wir das Programm schnell ganz überblicken und entsprechend im Detail bearbeiten können. Dadurch lernen wir die Probleme des Entwurfs und der Wartung großer Programme kaum kennen.
- Die Rechner früherer Jahre, mit denen wir unsere ersten Erfahrungen gesammelt haben, hatten einen sehr kleinen Speicher und waren langsam; wir mußten daher speicher- und laufzeiteffizient programmieren. Ähnliches gilt heute für die Rechner in Schulen und für die Home-Computer.
- Wir betrachten das laufende Programm als unser mit Abstand wichtigstes Ziel, nicht das Vorhandensein einer präzisen Dokumentation oder einer transparenten Programmstruktur. Auch unsere Vorgesetzten können die Frage, ob ein Programm schon läuft, wesentlich leichter beantworten als die nach Qualitäten, die sich bis heute nicht objektiv beurteilen lassen.

Aus diesen Gründen haben wir es uns zur Gewohnheit gemacht, möglichst schnell einen möglichst effizienten Code zu schreiben. Wir betrügen dabei uns selbst und unsere Auftraggeber: uns selbst, weil wir beim Testen einen unverhältnismäßig hohen Aufwand treiben, und unsere Auftraggeber, weil diese bei der späteren Programmwartung für unsere Eile und die Optimierung des Programms bezahlen. Daher gelten für die Optimierung des Programms die beiden Regeln von MICHAEL JACKSON:

Tue es nicht!
Wenn du es unbedingt tun mußt, tue es später!

Diese Regeln besagen, daß eine Optimierung in sehr vielen Fällen ganz unangebracht ist. Der Speicherplatz ist heute in praktischen Anwendungen meist kein Problem mehr, und wo er als Problem erscheint, wäre es oft viel billiger, zusätzliche Hardware anzuschaffen, als bei der Software Uhrmacherei zu betreiben. Hinsichtlich der Laufzeit ist bei Programmen oder Programmteilen, die nur selten gebraucht werden, die Optimierung unsinnig.

Als Preis der frühen Optimierung verzichten wir in der Regel auf die Transparenz des Programms und damit auf unsere Gewißheit, daß es korrekt ist. Wir sollten daher besser so vorgehen, wie es der obere Weg in Bild 3 zeigt:

Kriterien für gute Programmiersprachen und gute Programmierer

Wir haben nun die Voraussetzungen, einige Kriterien für den Vergleich von Programmiersprachen zu sammeln (Tabelle 2). Es wurde bereits zu Beginn des ersten Beitrages erwähnt, daß Programmiersprachen nicht nur eine technische, sondern auch eine psychologische Seite haben und daß ihre Bewertung daher unvermeidlich subjektiv ist. Allerdings ist es möglich, aufgrund empirischer Untersuchungen die Urteile zu überprüfen. Dies gilt besonders im Hinblick auf die ganze Fehlerproblematik. Wegweisend ist daher hier die Feststellung, daß es unser größtes Problem ist, ein Programm *korrekt* zu implementieren und auch bei Änderungen die Korrektheit zu erhalten. Beides soll mit möglichst geringem Aufwand erfolgen (aber nicht zu Lasten der Korrektheit). Eine gute Programmiersprache soll daher jede mögliche Hilfe zur Vermeidung und Erkennung von Fehlern leisten und im übrigen leicht erlernbar und ausreichend mächtig sein. Sie soll den Programmierer nicht zwingen, Übersetzungen vorzunehmen, die die Maschine intern besorgen könnte; andernfalls lenkt sie seine Aufmerksamkeit von den kritischen Punkten ab und behindert seine Arbeit.

FORTRAN hat in Tabelle 2 mit 6.1, 6.4 und 6.5 drei wichtige Pluspunkte, im übrigen aber fast nur Negativanzeigen.

Tabelle 2. Kriterien für den Vergleich von Programmiersprachen.

1. **Namen**
   - 1.1 Keine Beschränkungen bei der Länge von Namen
2. **Konstanten und Typen**
   - 2.1 Definition symbolischer Konstanten und Typen
   - 2.2 Aufbau komplexer Datentypen
   - 2.3 Strikte Typprüfung (Strong Typing)
   - 2.4 Unterstützung bei der Bildung abstrakter Datentypen
3. **Wertzuweisungen**
   - 3.1 Nebenwirkungen (Side Effects) ausgeschlossen oder scharf kontrolliert
4. **Ablauf**
   - 4.1 Sichere, geschlossene Konstrukte für den Ablauf
   - 4.2 Prozeduren mit nach Kommunikationsrichtung qualifizierten Parametern
   - 4.3 Keine allgemeine Sprunganweisung (GOTO), statt dessen Aussprung und Ausnahmebehandlung (Exception Handling)
   - 4.4 Höheres Sprachkonzept für die Prozeßkommunikation
5. **Stil und Umfang**
   - 5.1 Nach einem eingängigen Prinzip aufgebaut (nicht zusammengestoppelt)
   - 5.2 Überschaubar, leicht erlernbar
   - 5.3 An mathematischem Formalismus orientiert
6. **Übersetzer**
   - 6.1 Muß in nützlicher Frist auf den richtigen Maschinen verfügbar sein
   - 6.2 Benutzerfreundliche Fehlerbehandlung
   - 6.3 Einheitliche Bedienschnittstelle
   - 6.4 Separate Compilierung
   - 6.5 Hinreichend effizienter Zielcode

PASCAL erfüllt dagegen viele der Forderungen, schmerzlich vermißt werden 2.4 und 6.4, außerdem sind die Kriterien unter 4. nicht oder nicht befriedigend erfüllt. In MODULA-2 sind alle diese Mängel behoben, dafür hapert es bei 6.1; 6.3 bleibt ebenso ungelöst wie bei PASCAL. ADA schließlich wird, wenn Compiler in ausreichender Zahl verfügbar sind und diese 6.5 erfüllen, allen Kriterien (auch 6.3) genügen außer 5.2; an dieser Stelle müssen unsere Bemühungen ansetzen, ein didaktisches Konzept zu schaffen. Es sollte allerdings auch klar sein, daß die Schwierigkeiten bei der Einführung von ADA nicht so sehr bei ADA selbst liegen als vielmehr bei der mangelnden Basis, da vielfach Grundkenntnisse des Software Engineering fehlen.

Noch weniger klar lassen sich die *Eigenschaften eines guten Programmierers* angeben. Natürlich ist es vorteilhaft, wenn er sich auf eine gründliche Ausbildung stützt, abstrakt denken kann, komplizierte Zusammenhänge schnell versteht, ein gutes Gedächtnis hat usw. Hier sollen einige Merkmale genannt werden, die weniger leicht ins Auge springen:

- Das wichtigste Kennzeichen eines guten Programmierers ist seine *Dis-*

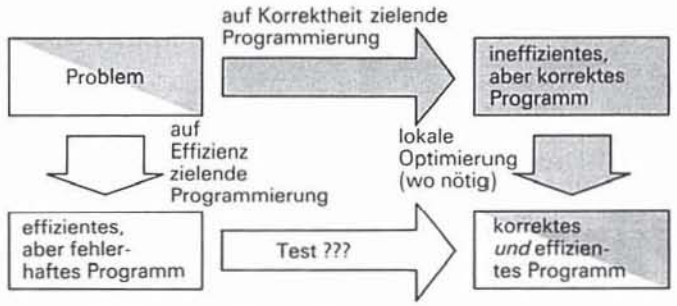


Bild 3. Geeignete und ungeeignete Vorgehensweise zur Programmoptimierung.

ziplin. Dank ihr widersteht er dem Drang, so schnell wie möglich zu codieren und zu testen, legt statt dessen seine Programme gründlich und sauber an und dokumentiert *vor* der Codierung. Unzulänglichkeiten der Programmiersprache kann der Programmierer durch Disziplin teilweise ausgleichen, indem er, wo nötig, «so tut, als ob», also etwa in FORTRAN GOTO nur so einsetzt, wie es in MODULA möglich ist, usw. Die Disziplin veranlaßt ihn auch, sich an Regeln und Normen zu halten und seine Programme nach dem Problem zu formen und nicht nach sich selbst («ego-free-programming»). Schließlich schützt ihn die Disziplin vor Unbeständigkeit und Schwäche gegenüber wechselnden Wünschen aus der Umgebung (er kann *NEIN* sagen).

- Ein guter Programmierer hat ein intuitives Verständnis dafür, was *Komplexität* bedeutet. Der Unterschied zwischen 10 Zeilen und 10 000 Zeilen besteht für ihn nicht nur in einem Faktor tausend. Er weiß, welche *Qualität* des Problems durch die Quantität hinzukommt, und er richtet sich auf die damit verbundenen Schwierigkeiten ein.
- Immer wieder wird der Programmierer von einer soliden Grundlage in Mathematik profitieren (Algebra, Automatentheorie usw.); fehlt dieser Teil der Ausbildung, so läßt er sich neben der Praxis kaum noch nachholen.
- Um mit einer sich schnell entwickelnden Technik Schritt zu halten, ist ein guter Programmierer *flexibel* und *lernfreudig*; er ist *kooperativ* und auch bereit, seine Arbeit der *Kritik* seiner Kollegen auszusetzen.

Es sollte nach allen Aussagen dieses Berichts klar sein, daß Programmieren einerseits keine «schwarze Kunst» ist, anderseits aber auch kein Kinderspiel, das jeder Laie nach ein paar Wochen in einem Programmierkurs – oder schlimmer noch nach einigen privaten Versuchen am Home-Computer – beherrscht. Programmieren ist ein «Kopfwerk», das man solide erlernen und anwenden sollte. Daran ändert auch die Tatsache nichts, daß heute in der Praxis viel Pfusch üblich ist.

Teilgebiete, die in diesem Aufsatz nicht behandelt sind

Dieser Beitrag kann nicht flächendekkend sein. Zu den vielen wichtigen Teilgebieten, die nicht behandelt werden, zählen

- Spezialsprachen für bestimmte Anwendungen, zum Beispiel numerisch gesteuerte Werkzeugmaschinen
- die Sprachen aus dem Gebiet der künstlichen Intelligenz (LISP, PROLOG)
- die Diskussion über grafische Sprachen und damit der Hinweis, daß Flußdiagramme überholt sind und zumindest NASSI-SHNEIDERMAN-Diagramme benutzt werden sollten [24]
- eine Erweiterung des Programmiersprachenbegriffs in Richtung auf Sprachen, die zur Spezifikation und zum Entwurf verwendet werden [25]
- axiomatische Spezifikation und transformatorische Programmentwicklung, Verifikation und Validation von Programmen
- Methoden zur Definition von Programmiersprachen (über BNF hinaus)
- die Bedeutung nichtprozeduraler Sprachen und ihr Zusammenhang mit neueren Entwicklungen weg von der VON-NEUMANN-Maschine [26]
- die Grundzüge der Sprachentheorie, vor allem die Einteilung in reguläre, kontextfreie und kontextsensitive Sprachen und einige wichtige Lehren der Theorie.

Außerdem sind natürlich viele bekannte Programmiersprachen nicht oder nur kurz erwähnt, wie beispielsweise PL/I, PEARL, C, BLISS, JOVIAL, CORAL. [13]

Literatur

17 Wirth N. (1982): Programming in MODULA-2. Springer-Verlag, New York, Heidelberg, Berlin.
18 Kreifelts Th., Schnupp P. (1983): UNIX, Konzepte und Anwendungen. Bericht Nr. 12 des German Chapter of the ACM, B. G. Teubner, Stuttgart.
19 Ledgard H./DoD (1981): Ada – An Introduction – Ada Reference Manual. Springer-Verlag, New York, Heidelberg, Berlin. (Vermutlich demnächst in neuer Auflage, etwa 1983.)
20 Hibbard P., Hisgen A., Rosenberg J., Shaw M., Sherman M. (1983): Studies in Ada Style, Springer-Verlag, New York, Heidelberg, Berlin.
21 Booch G. (1983): Software Engineering with Ada, The Benjamin/Cummings Publishing Co., Inc., Menlo Park, California.
22 Gries D. (1976): Some comments on programming language design. In Schneider H. J., Nagl M. (Hrsg.): Programmiersprachen, 4. Fachtagung der GI, Erlangen, März 1976, Springer, pp. 235–252.
23 Schärf J. (1977): BASIC für Anfänger, R. Oldenbourg Verlag, Wien, München, 5. Auflage.
24 Schnupp P. (1974): Struktogramme – eine neue Methode der Systemplanung, ONLINE, November 1974, S. 734–741.
25 Ludewig J. (1983): Bericht von der 54. SGA-Tagung «Spezifikation von Realzeitsystemen – Konzepte, Lösungen, Erfahrungen». SGA-Bulletin, 1983, Heft 2, S. 9 bis 18.
26 Leavenworth B. M., Sammet J. (1974): An overview of nonprocedural languages, Sigplan Notices, 9,4, 1 bis 12.

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 kürzlich im Bibliographischen Institut in Mannheim erschienen.