Programme für Rechenanlagen werden in formalen Sprachen abgefaßt, die stets einen Kompromiß darstellen zwischen den Wünschen und den Fähigkeiten des Programmierers einerseits und den technischen Möglichkeiten des Rechners anderseits. Der dreiteilige Beitrag erläutert wichtige Aspekte der Programmierung und stellt einige der bekanntesten Sprachen anhand ihrer besonderen Merkmale vor.

Sprachen für die Programmierung – eine Übersicht

Von Jochen Ludewig

Kein Mensch weiß, wie viele Programmiersprachen es gibt. Zeitweise gehörte es geradezu zum guten Ruf eines Informatikers, seine eigene Sprache zu erfinden. Auch wenn man von solchen Exoten absieht, bleiben mehrere Dutzend Sprachen übrig (mit Dialekten einige hundert), die tatsächlich verwendet werden. Daher wäre es heute vermessen, einen vollständigen Überblick der Sprachen anzustreben. Bestenfalls kann man die wichtigsten Grundelemente und Merkmale der gängigsten Programmiersprachen herausarbeiten und ihre Bedeutung diskutieren.

Programmiersprachen schlagen die Brücke zwischen dem menschlichen Verstand und der elektronischen Maschine, auf der die Programme ausgeführt werden. Darum ist es bei diesem Thema notwendig, sich mit beiden Aspekten, also der Technik und dem Menschen, zu befassen. Es wird daher versucht, beide Gesichtspunkte und ihre Wechselwirkung zum Ausdruck zu bringen. Der menschliche Aspekt bedingt, daß es bei den Wertungen keine absolute Wahrheit geben kann.

Mit Ausnahme weniger geschichtlicher oder wissenschaftlicher Hinweise ist das Thema dieses Beitrags eingeschränkt auf weitverbreitete und relativ allgemein verwendbare Programmiersprachen *(general purpose programming languages)* für konventionelle Rechner, die automatisch ausführbar oder in eine ausführbare Form übersetzbar sind [1, 2]. Damit sind sehr spezielle Sprachen ausgeschlossen. Solche *problemorientierten Sprachen* finden Anwendung bei der Programmierung numerisch gesteuerter Werkzeugmaschinen, für die Berechnung elektrischer Netzwerke oder zur Simulation diskreter oder kontinuierlicher Systeme. Gewisse wissenschaftliche Disziplinen, vor allem die «Künstliche Intelligenz», haben sich ebenfalls ihre Spezialsprachen geschaffen.

Bei der Schöpfung der Programmiersprachen, über die hier gesprochen werden soll, hat es verschiedene Traditionen gegeben, durch die sich gewisse «Familien» gebildet haben. Bild 1 soll sehr grob die wichtigsten Beziehungen aufzeigen.

Der Beitrag folgt im wesentlichen der Einteilung in die folgenden Gruppierungen:

- Assembler-Sprachen als unterste Stufe der Programmierung
- FORTRAN als erste, noch heute in vielen Gebieten dominierende höhere Programmiersprache für technisch-wissenschaftliche Probleme
- COBOL als typische Sprache der kommerziellen Programmierung
- die Familie der ALGOL-ähnlichen (oder *blockorientierten*) Sprachen, zu denen neben ALGOL 60 und ALGOL 68 auch SIMULA, PASCAL und mit Einschränkungen PL/1 gehören
- die Familie der auf den blockorientierten Sprachen aufbauenden modulorientierten Sprachen, bestehend aus SIMULA, MODULA und ADA
- die Familie der Dialogsprachen, vertreten durch BASIC

Natürliche und formale Sprachen

Natürliche Sprachen bezeichnen zunächst das *Gesprochene*. In diesem Sinne ist etwa das Schweizerdeutsch eine typische natürliche Sprache. Mit der Erfindung der Aufzeichnung sprachlicher Mitteilung durch Schrift und durch andere Darstellungen entstand ein abstrakterer Sprachbegriff, der alle denkbaren Repräsentationen einschließt. Sprache ist damit *jede Form der Kommunikation*.

Diesem Beitrag soll der folgende Sprachbegriff zugrunde gelegt werden: Eine *Sprache* besteht aus der Gesamtheit aller darin möglichen Aussagen und ihrer Bedeutungen. Sie ist definiert durch ein System von Regeln, die sogenannte *Grammatik*. Die Grammatik besteht aus zwei Teilen:

- Regeln darüber, welche Sätze in der Sprache möglich (das heißt zulässig)

Die Programmiersprachen schlagen die Brücke zwischen dem Verstand des Programmierers und dem Computer, der die Programme abarbeitet. (HR. Bramaz)

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

sind (dieser Teil der Grammatik heißt *Syntax*)

- Regeln, die den zulässigen Sätzen Bedeutungen zuordnen (diese werden als *Semantik* bezeichnet)

Die Unterscheidung soll an einigen einfachen Beispielen erläutert werden. Dabei soll uns die deutsche (Schrift-)Sprache als Beispiel dienen.

- «Regen gestern total Programmiersprachen» ist offenbar kein zulässiger Satz der deutschen Sprache. Es stellt sich daher nicht die Frage nach seiner Bedeutung, denn die Semantik ist nur für korrekte Sätze definiert
- «This is a report on programming languages» ist ebenfalls kein zulässiger Satz der deutschen Sprache.
- «Diese Seite hat zehn Zeilen» ist ein syntaktisch korrekter Satz. Er hat damit auch eine Semantik; wir können den Satz verstehen und ihm zustimmen oder ihn bestreiten. In diesem speziellen Fall ist der Satz *inhaltlich* falsch.

Bei den natürlichen Sprachen sind Syntax und Semantik nicht präzise festgelegt, sondern durch Erfahrungen bei der Kommunikation erworben. Dabei sind unterschiedliche Vorstellungen unvermeidlich. So sind vermutlich die Meinungen geteilt, ob der Satz «Software-Engineering ist in» zur deutschen Sprache gehört. Wo dies klar ist, sind wir keineswegs sicher über die Bedeutung. «Das Wetter ist schön» bedeutet für jeden etwas anderes, und selbst scheinbar quantifizierende Aussagen wie «Dies ist der größte Wasserfall der Welt» sind unklar, weil sich die Größe in der Höhe, der Breite, der Durchflußmenge oder «einfach» in unserer Empfindung ausdrücken kann.

Künstliche Sprachen lösen teilweise die Probleme, da sie auf eine wohldefinierte Syntax gegründet sind. Esperanto als Vertreter solcher Sprachen ist daher leichter erlernbar als etwa Deutsch, hat damit aber noch keine definierte Se-

mantik, denn die Bedeutung ist nur durch Übersetzungen in natürliche Sprachen definiert. Außerdem ist die Syntax sehr wahrscheinlich noch nicht frei von Unklarheiten.
Erst *formale Sprachen* beseitigen (prinzipiell) diese Schwierigkeiten. Sie sind formal definiert, das heißt durch einen mathematischen Formalismus. Im Idealfall deckt die Definition sowohl die Syntax als auch die Semantik ab; praktisch ist sie meist auf die Syntax beschränkt, die Semantik ist in natürlicher Sprache beschrieben.

Definition formaler Sprachen

Als einfaches Beispiel für eine Sprachdefinition seien hier einige Elemente von arithmetischen Ausdrücken beschrieben.

Ziffer ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

Diese *Produktion* bedeutet folgendes: Eine Ziffer kann eine Null, eine Eins, . . ., eine Acht oder eine Neun sein. Das Symbol «::=» bedeutet «ist definiert als», der senkrechte Strich trennt Alternativen voneinander, der Punkt beendet die Produktion. «1» und «7» sind also Ziffern, «x» und «22» sind keine.
Nun soll eine Zahl aus beliebig vielen Ziffern definiert werden:

Zahl ::= Ziffer | Ziffer Ziffer | Ziffer Ziffer Ziffer | . . .

Natürlich können nicht alle Möglichkeiten aufgeführt werden, denn es gibt ja unendlich viele. Hier hilft das Prinzip der *rekursiven Definition:*

Zahl ::= Ziffer | Ziffer Zahl.

Danach ist etwa die Zahl «5» durch die erste Alternative ausgedrückt, die Zahl «350» dagegen durch zweimalige rekursive Anwendung der zweiten und abschließend der ersten Alternative («350» ist Ziffer «3», gefolgt von der Zahl «50», worin «50» die Ziffer «5» ist, gefolgt von der Zahl «0», worin die Zahl «0» die Ziffer «0» ist).
Schließlich kann auf die gleiche Weise eine Summe definiert werden als

Summe ::= Zahl | Zahl + Summe.

Damit ist unzweideutig festgelegt, daß «7», «0000+0», «13+555+17» Summen sind, «+88», «Pi+1» oder «12/4» dagegen keine Summen sind (oder, wie es in der Informatik heißt, nicht *zu den Wörtern der Sprache gehören).* Man beachte, daß nach dieser Definition Leerstellen (Blanks) in den Summen (zum Beispiel «3 + 9») nicht zulässig sind; die dazu notwendige Erweiterung ist aber einfach.
«Summe», «Zahl» und «Ziffer» werden als *syntaktische Variablen* bezeichnet, die schließlich durch sogenannte *Terminalzeichen* (hier «+» und die Ziffern) ersetzt werden. Eine Summe kann nun mit Hilfe der drei angegebenen Produktionen erzeugt werden, indem, mit dem Wort «Summe» beginnend, die Produktionen angewendet werden, bis alle syntaktischen Variablen ersetzt sind. Um diese und die Hilfssymbole (zum Beispiel «::=») nicht mit den Terminalzeichen zu verwechseln, wird zum Beispiel durch einen anderen Schrifttyp die Zugehörigkeit gekennzeichnet.
Das Beispiel oben verwendet die sogenannte *«Backus-Naur-Form»* oder *BNF,* in der ALGOL 60 und viele andere Sprachen definiert wurden. Diese Sprache zur Definition von Sprachen (daher: *Meta-Sprache)* ist heute weit verbreitet [3]. Für die formale Definition der Semantik, die hier nicht behandelt werden soll, gibt es komplizierte Meta-Sprachen.
Bei vielen Programmiersprachen fehlt eine formale Definition. Dazu wird vielfach die Ansicht vertreten, eine formale Definition sei ohnehin zu kompliziert für den typischen Anwender. Hier liegt ein Mißverständnis vor. Die Definition ist keine Gebrauchsanleitung (dazu gibt es Lehrbücher), sondern sie dient dem Anwender so, wie eine Brückenberechnung dem Benutzer dient: Sie schützt ihn vor dem Absturz, auch wenn er sie nicht versteht. Soweit eine Sprache formal definiert ist, gibt es keine (unbeabsichtigten) Dialekte, so daß Programmierer leichter auf verschiedenen Rechnern arbeiten können und Programme übertragbar sind.

Algorithmen

Ein *Algorithmus* ist eine präzise Anweisung, wie eine Berechnung durchgeführt werden soll. Jeder kennt einfache Beispiele, so etwa für die Division zweier natürlicher (das heißt positiver und ganzer) Zahlen mit Rest:

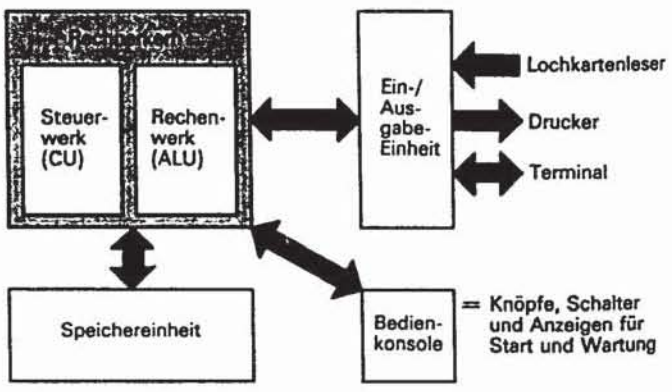

Bild 2. Struktur eines einfachen Rechners.

◄ Bild 1: Ahnentafel der bekanntesten Programmiersprachen. Die Jahreszahlen bezeichnen ungefähr das Geburtsjahr der Sprache. Die Pfeile kennzeichnen die wichtigsten Einflüsse.

Ist der Dividend kleiner als der Divisor, so ist der Quotient gleich Null. Andernfalls ist der Quotient um eins größer als der Quotient, der sich ergibt, wenn die gleiche Rechnung mit einem um den Divisor verminderten Dividenden ausgeführt wird.
Natürlich sind formale Sprachen besonders geeignet für die präzise Beschreibung einer Berechnung. Daher werden sie vielfach auch als *algorithmische Sprachen* bezeichnet. Vergleiche dazu die Formulierung des oben verbal beschriebenen Algorithmus in der Programmiersprache PASCAL:

```
FUNCTION Quotient (Dividend, Divisor  integer): integer;
BEGIN
  IF Dividend < Divisor
  THEN Quotient  = 0
  ELSE Quotient  = Quotient (Dividend - Divisor, Divisor) + 1
END.
```

Im Zusammenhang mit Programmen bezeichnet das Wort «Algorithmus» den abstrakten Kern, also die Semantik.
Erläuterung zur Funktion Quotient:
Eine solche Funktion ist nur als Teil eines größeren Programmes brauchbar, in dem sie beliebig oft *aufgerufen* wird, zum Beispiel durch einen Befehl

a:= Quotient (21,4)

Offenbar erhält a den Wert 5. Die Funktion ist rekursiv definiert, das heißt, im Zuge ihrer Ausführung ruft sie sich selbst auf. (Das Beispiel ist natürlich praktisch nicht sinnvoll.)

Anfänge der Programmiersprachen

Ähnlich wie LEONARDO DA VINCI viele technische Entwicklungen vorausgesehen hat, ohne sie tatsächlich zu beeinflussen, gibt es in der Informatik Arbeiten von SCHICKARD, LEIBNIZ, PASCAL, BABBAGE und anderen, die im nachhinein in ihrer Bedeutung gewürdigt werden können, aber keine praktischen Auswirkungen auf die Entwicklung hatten [4]. Selbst das vom Computerpionier CONRAD ZUSE in den vierziger Jahren entworfene «Plankalkül», das wesentliche Elemente moderner Programmiersprachen enthält, ist nicht wirksam geworden. Eher schon spielten einfache Techniken der Informationsverarbeitung aus dem 19. Jh. eine Rolle (Lochstreifensteuerung für Webstühle, die noch heute verwendete Lochkarte von HOLLERITH).
Die ersten, während des Zweiten Weltkriegs in den USA und in Deutschland gebauten Rechner, die aus Relais aufgebaut waren, wurden durch starre, in Kinofilm gelochte Programme gesteuert. Jedes Loch im Streifen setzte eine bestimmte Rechenoperation in Gang. Die «Programmiersprache» war also syntaktisch durch die möglichen Lochungen, semantisch durch die davon ausgelösten Operationen definiert.

Die Von-Neumann-Maschine

Die Entwicklung der Programmiersprachen im heutigen Sinne begann mit den programmierbaren Rechenautomaten, wie sie kurz nach dem Krieg in den USA zum erstenmal zur Verfügung standen. JOHN VON NEUMANN formulierte als wesentliche Merkmale solcher Rechner *(Von-Neumann-Maschine):*

- Programme werden im selben Speicher abgelegt wie die Daten, mit denen gerechnet wird. Damit besteht die Möglichkeit, Programme wie Daten zu behandeln, insbesondere sie in andere Sprachen zu übersetzen.
- Die Maschine kennt mindestens einen Befehl, um das Programm in Abhängigkeit von Rechenergebnissen an anderer Stelle fortzusetzen. Diese sogenannten *bedingten Anweisungen* erlauben es, den Programmablauf durch die Daten und Zwischenergebnisse steuern zu lassen.

Ein (aus heutiger Sicht sehr einfacher) Rechner [3] besteht aus Rechnerkern, Ein-/Ausgabeeinheit, Speichereinheit und Bedienungskonsole (Bild 2). Der Rechnerkern zerfällt in Steuerwerk und Rechenwerk; letzteres enthält die *Register,* in denen Rechnungen ausgeführt werden können. Im folgenden Beispiel gibt es nur ein einziges Register zum Rechnen, den sogenannten *Akkumulator.* Dieser kann ein *Wort* speichern, das 32 Bits enthält, in anderen Fällen typisch auch 16, 24 oder 64 Bits. Auch der Speicher ist in Wörtern organisiert, dort kann allerdings nicht gerechnet werden, es ist immer der Umweg über den Akkumulator erforderlich. Soll also zum Beispiel der Wert einer Zahl, die in Speicherzelle x steht, um eins erhöht werden, so läuft dies wie folgt ab:
Lade Wert aus Speicherzelle x in den Akkumulator;
Addiere im Akkumulator die Zahl 1;
Speichere Inhalt des Akkumulators nach Speicherzelle x.
Mit x = 0011 1100 1000 1010 (Zwischenräume nur zur Übersicht) kann dies in der Maschinensprache etwa wie folgt aussehen:

```
0000 1010 0000 0000 0011 1100 1000 1010
0000 0011 0000 0000 0000 0000 0000 0001
0000 1011 0000 0000 0011 1100 1000 1010
```

Darin beschreibt die linke Hälfte des Wortes jeweils die Operation, die rechte Hälfte den Operanden oder die Adresse. Das Beispiel deutet natürlich die Möglichkeiten nur ganz vage an.
Die *binäre* Darstellung oben, mit der der Rechner tatsächlich arbeitet, ist für den Menschen absolut ungeeignet. Eine erste Verbesserung läßt sich durch die Zusammenfassung von je vier Bits zu einer *Tetrade* erreichen. Diese werden mit den Zeichen 0, 1, 2, ..., 9, A, B, C, D, E, F, den *Sedezimalziffern* bezeichnet. Die drei Befehle oben lauten dann

```
0A00 3C8A
0300 0001
0B00 3C8A
```

Assembler

Ein wesentlicher Fortschritt wird nun durch die Verwendung von Symbolen erreicht, die jeweils an die Stelle eines bestimmten Codes treten, zum Beispiel LDA für «Lade Akkumulator», AAD für «Addiere Adreßteil» und SPA für «Speichere Akkumulatorinhalt».

```
LDA    3C8A
AAD    0001
SPA    3C8A
```

Während der Sedezimalcode oben meist von der Hardware direkt in die entsprechenden Bitmuster umgesetzt werden kann, ist nun erstmals eine *Übersetzung* notwendig. Dazu dient ein besonderes Programm, der *Assembler* (von to assemble = zusammenbauen). Die Übersetzung ist einfach, weil

- die Angaben in bestimmten Stellen der Eingabe stehen *(formatgebunden),* also bei Lochkarten immer in derselben Spalte
- in der Regel jede Zeile in eine feste Zahl von Rechnerwörtern abgebildet wird (in diesem Beispiel jeweils 1 zu 1)
- die Symbole fest auf die zugeordneten Codes abgebildet werden können

Betrachten wir ein zweites, etwas komplizierteres Beispiel. Von einer gegebenen Zahl soll die Fakultät berechnet werden, also das Produkt der Zahl mit allen kleineren Zahlen bis 1 (Beispiel: Fakultät von 1 ist 1, von 2 2, von 4 24 usw.).
Dies Programm lautet in unserer Beispiel-Assemblersprache wie folgt (die Adresse in der ersten Spalte steht nur zur Information, sie ist nicht Teil des Programms):

Adresse	Operator	Operand	Bedeutung des Operators
1F21	LAT	0001	Lade Adreßteil
1F22	SPA	4B03	Speichere Akku-Inhalt
1F23	LDA	4B02	Lade Akkumulator
1F24	SKG	1F2B	Sprung wenn kleiner oder gleich
1F25	MUL	4B03	Multipliziere
1F26	SPA	4B03	Speichere Akku-Inhalt
1F27	LDA	4B02	Lade Akkumulator
1F28	SAT	0001	Subtrahiere Adreßteil
1F29	SPA	4B02	Speichere Akku-Inhalt
1F2A	SGR	1F25	Sprung wenn größer
1F2B	...	...	
	...	...	
4B02	0000	0007	
4B03	0000	0000	

Symbolische Adressierung

Ein schwerer Mangel dieser Darstellung ist die Verwendung *absoluter Adressen.* Wird im Programm oben etwa ein Befehl eingeschoben oder wird es an einer anderen Stelle des Speichers geladen, so müssen die beiden Sprungadressen von Hand geändert werden. Hier schaffen die *symbolischen Adressen* Abhilfe.
Speicherplätze für Daten werden an beliebiger Stelle definiert, Befehle werden mit Marken versehen. Die Symbole werden durch das Gleichheitszeichen von den Operationscodes unterschieden. Mit symbolischen Adressen lautet das Programm nun:

```
        LAT     1
        SPA     FAKUL
        LDA     ARGUM
        SKG     ENDE
LOOP =  MUL     FAKUL
        SPA     FAKUL
        LDA     ARGUM
        SAT     1
        SPA     ARGUM
        SGR     LOOP
ENDE =  ...     ...
        ...     ...
ARGUM =         7
FAKUL =         0
```

Damit ist etwa die Sprachebene einfacher Assembler [5] erreicht. Diese Sprache ist für den Menschen viel besser verwendbar als die Maschinensprache, weil

- sich die Operationssymbole viel leichter erlernen und lesen lassen
- Programmänderungen keine unüberschaubaren Effekte auf Adressen haben
- freigewählte Adreßsymbole erklärend wirken

Dennoch ist die Assemblersprache stark *maschinenabhängig;* für jede Operation der Maschine gibt es einen Befehl, so daß das Programm in der Regel nicht auf Rechnern eines anderen Typs laufen kann.
Das Programm enthält (ab Marke LOOP) einen Abschnitt, der im allgemeinen wiederholt durchlaufen wird, eine sogenannte *Laufschleife.* Diese realisiert die *Iteration,* ein fundamentales Konzept der Programmierung. Charakteristisch sind dabei

- die Initialisierung vor Eintritt in die Schleife
- die Prüfung auf die Abbruchbedingung (der *bedingte Aussprung)*
- der Rücksprung zum Anfang der Schleife

Da das Programm genau wie die Daten im Speicher liegt, ist es möglich, daß es sich selbst modifiziert, also etwa einen Befehl durch einen anderen ersetzt.
Solche Tricks waren früher sehr beliebt, gelten inzwischen jedoch zu Recht als Todsünde, weil sich ein selbstmodifizierendes Programm jeder Kontrolle entzieht. Daher wird in den «höheren» Programmiersprachen diese Möglichkeit ausgeschlossen.
Assemblersprachen waren bis zum Erscheinen von FORTRAN die einzigen Programmiersprachen überhaupt. Auch danach haben sie – zum Teil bis heute – große Bedeutung behalten. Sie gestatten es wie keine andere Programmiersprache, mit allen Tricks und Finessen zu arbeiten und extreme Resultate hinsichtlich des Speicherbedarfs oder der Geschwindigkeit zu erzielen. Anderseits sind Assemblerprogramme aufwendig zu erstellen und noch aufwendiger zu korrigieren und zu verändern. Schließlich sind sie nicht *portabel,* lassen sich also nicht auf andere Maschinen übertragen. Da heute in der Regel die Elektronik (Hardware) billiger ist als die darauf laufenden Programme (Software), hat das Argument der Effizienz seine Bedeutung verloren. Der Einsatz von Assemblersprachen ist nur noch gerechtfertigt, wo aus technischen Gründen keine anderen Lösungen möglich sind. Solche Fälle sind allerdings selten und vielfach nur aus Bequemlichkeit vorgeschoben.

Virtuelle Maschinen

Der Programmierer kann nun so tun, als ob die Maschine selbst die Assemblersprache verstünde, denn er braucht sich mit der Maschinensprache nicht mehr zu befassen. Er arbeitet folglich in seiner Vorstellung nicht mit der realen, sondern mit einer *virtuellen Maschine.* Eine ähnliche Situation besteht für uns, wenn wir mit jemandem korrespondieren, der unsere Sprache nicht beherrscht. Die Tatsache, daß unser Brief zunächst übersetzt und dann erst bearbeitet wird und daß die Antwort umgekehrt ebenfalls übersetzt werden muß, können wir ignorieren. Übersetzer und Bearbeiter bilden also für uns einen virtuellen Bearbeiter.
Definition: Eine virtuelle Maschine entsteht aus einer realen Maschine, indem letztere mit Hilfsmitteln ausgestattet wird, die die Programmierung in anderen Sprachen als der eigentlichen Maschinensprache ermöglichen.
Auf jedem Rechner sind heute solche virtuellen Maschinen verfügbar, bei großen Systemen oft mehr als ein halbes Dutzend. Meist bauen sie aufeinander auf, das heißt, neue virtuelle Maschinen werden mit Hilfe bereits vorhandener realisiert.
Eine gute virtuelle Maschine ist «dicht», das heißt, sie gibt in keinem Fall den Blick auf die eigentliche (oder eine untergeordnete) Maschine frei. Nehmen wir an, daß es bei der Multiplikation zu einem Überlauf kommt, also das Produkt nicht mehr in das 32-Bit-Wort paßt. Die virtuelle Maschine darf nun nicht melden

```
MULTIPLICATION OVERFLOW   AT LOCATION 0007
```

sondern sollte sich auf die Symbole im Programm beziehen, etwa durch

```
MULTIPLICATION OVERFLOW 0 STEPS AFTER LABEL "LOOP"
```

FORTRAN

In den Assemblersprachen ist die Programmierung mathematischer Ausdrücke sehr mühsam. So benötigt etwa die Formel

$$y = c_1 (2x - 1) + c_2 (3x^2 - 2x)$$

mehr als zehn Zeilen Assembler. Da die ersten Anwendungen der Computer ausschließlich dem technisch-wissenschaftlichen Rechnen dienten, wo solche Formeln eine wichtige Rolle spielen, begann man in den fünfziger Jahren, Verfahren zu einer automatischen Übersetzung ganzer Formeln zu entwikkeln. 1954 wurde bei IBM eine Sprache zu diesem Zweck vorgeschlagen, die den Namen FORTRAN (aus FORmula TRANslating system) erhielt [6, 7]. FORTRAN war also eine Art Superassembler. So erklären sich einige Merkmale, die FORTRAN in seinen vielen Dialekten bis heute bewahrt hat:

- Die Zeilenstruktur spiegelt noch wie beim Assembler die Anordnung im Speicher, die Syntax ist formatgebunden, allerdings weniger eng.
- Schnelle Codierung geht vor «moderneren» Zielen wie Vermeidung von Fehlern oder leicht durchschaubaren Programmstrukturen.

Das Programm oben lautet nun in FORTRAN sehr einfach

```
      NFAK = 1
      DO 50 I = 1, N
50    NFAK = NFAK * I
```

NFAK, N und I sind *Variablen.* Diese erkennt der Übersetzer und ordnet ihnen eine Speicheradresse zu. Das Gleichheitszeichen ist, strenggenommen, irreführend, denn mathematisch ist die Aussage NFAK = NFAK * I nur korrekt, wenn I = 1 ist. Tatsächlich handelt es sich aber um eine *Wertzuweisung* (Assignment); der Ausdruck auf der rechten Seite (NFAK * I) wird berechnet und anschließend der links genannten Variablen zugewiesen. NFAK ändert also den Wert (daher «Variable»).
In jüngeren Programmiersprachen wird für die Wertzuweisung meist ein anderes Symbol verwendet, häufig «:=».
«50» stellt hier keinen Zahlenwert dar,

sondern eine Marke, auf die in der *Laufanweisung* «DO ...» Bezug genommen wird. Die Laufschleife ist nun sehr viel leichter erkennbar: Auf die Initialisierung (NFAK = 1) folgt die Angabe, welcher Bereich (bis Marke 50) durchlaufen werden soll, wobei ein *Schleifenzähler* die Werte 1 bis N durchläuft.
Die Formulierung oben stellt natürlich noch kein vollständiges Programm dar, es fehlen die Angaben zur Ein- und Ausgabe. Außerdem ist es vorteilhaft, dem Übersetzer mitzuteilen, wie das ganze Programm heißen soll und wo es endet (PROGRAM und END). Für die bessere Lesbarkeit und Verständlichkeit fügt man Kommentarzeilen ein («C» in der ersten Spalte).
Unten ist ein komplettes Programm angegeben. Es ist gegenüber der Aufgabe noch erweitert, indem die wiederholte Eingabe von Zahlen vorgesehen wird. Der Abbruch erfolgt durch die Eingabe einer Null (Fakultät 0 kann nicht mehr berechnet werden). Die vorher genannte Schleife ist also in eine äußere Schleife *geschachtelt*. Die Formatanweisungen dienen der Ein- und Ausgabe, sie können hier nicht näher erläutert werden.

```
      PROGRAM FAKULTAET
C                          LIEST EINE ZAHL EIN UND DRUCKT DIE FAKULTAET AUS
   10 WRITE (6, 20)
   20 FORMAT (1H0, 22HBITTE ZAHL EINTIPPEN  )
      READ (5, 30) N
   30 FORMAT (I4)
      IF (N) 90, 90, 40
C
   40 NFAK = 1
      DO 50 I = 1, N
      NFAK = NFAK * I
   50 CONTINUE
C
      WRITE (6, 60) N, NFAK
   60 FORMAT (1H0 DIE FAKULTAET VON , I4, 5H IST , I9)
      GOTO 10
C
   90 STOP
      END
```

Die Marke 50 wurde hier von der Wertzuweisung abgetrennt und statt dessen vor ein (wirkungsloses) CONTINUE gesetzt. Dies ist in der Regel vorteilhaft, weil nun die Laufanweisung und das Schleifenende die eigentlichen Anweisungen echt umfassen. Soll zum Beispiel (bei veränderter Aufgabe) nach der Multiplikation noch eine weitere Anweisung in der Schleife liegen, so kann diese ohne Verschiebung der Marke eingefügt werden.
Im Beispiel oben sind die Marken in aufsteigender Ordnung verwendet. Das verbessert die Übersicht, ist aber keineswegs durch die Sprache erzwungen.
Ein Lauf des Programms kann wie folgt aussehen:

```
BITTE ZAHL EINTIPPEN
5
DIE FAKULTAET VON     5  IST          120

BITTE ZAHL EINTIPPEN
8
DIE FAKULTAET VON     8  IST        40320

BITTE ZAHL EINTIPPEN.
0
```

Wie man sieht, ist die Aufgabe des Übersetzers nun sehr viel komplizierter geworden. In der Regel wird das Programm mehrfach durchgearbeitet, bis es in Maschinencode transformiert ist. Man spricht daher nicht mehr von einem Assembler, sondern von einem *Compiler* (to compile = zusammentragen).
FORTRAN entwickelte sich rasch zu FORTRAN II und bis Mitte der sechziger Jahre zu FORTRAN IV. Einige Neuerungen jüngerer Sprachen wurden schließlich in FORTRAN-77 aufgenommen. Außerdem entwickelten sich viele Spezialversionen, so für Echtzeitanwendungen (Real-time-FORTRAN) und für die Simulation (GPSS-FORTRAN). FORTRAN IV und FORTRAN-77 wurden in Amerika standardisiert, wodurch die Unterschiede zwischen den Dialekten auf verschiedenen Rechnern vermindert wurden. Die folgenden Feststellungen gelten vor allem für das älteste brauchbare FORTRAN, nämlich FORTRAN II.

- Der Schwerpunkt liegt bei Arithmetik, die Sprache ist ungeeignet für Stringoperationen (Textbehandlung) und logische Operationen.
- Die Sprache ist im Gegensatz zu Assemblersprachen zu einem großen Teil maschinenunabhängig, da typisch maschinenabhängige Operationen (Zugriffe auf Register, Veränderungen des Maschinencodes) nicht mehr möglich sind.
- Es gibt zum erstenmal einfache Datenstrukturen (ARRAY).
- Teilaufgaben können als Unterprogramme (SUBROUTINE) abgetrennt werden.
- Für die Kommunikation zwischen Teilprogrammen gibt es neben den Parametern einen einfachen, sehr unsicheren Mechanismus, die COMMON-Bereiche.

FORTRAN hat die Programmiertechnik grundsätzlich gewandelt und die Rechner auch denjenigen zugänglich gemacht, die nicht hauptberuflich programmieren, so zum Beispiel den Physikern, die seitdem bevorzugt mit dieser Sprache arbeiten. Im Vergleich mit modernen Sprachen hat FORTRAN erhebliche Mängel, vor allem hinsichtlich der Sicherheit. Oft steht allerdings die große Menge von Programmen, die im Laufe der vergangenen 25 Jahre in FORTRAN erstellt wurden, dem Übergang auf eine andere Sprache entgegen.

COBOL

Die Geschichte von COBOL

Unter kommerziellen Programmen versteht man solche, die im kaufmännisch-administrativen Bereich eingesetzt werden. Beispiele sind Programme zur Verwaltung großer Dateien, wie sie etwa bei Verwaltungen oder Versicherungsgesellschaften bestehen. Es handelt sich also hier in der Regel nicht um das Problem, eine komplizierte Berechnung zu automatisieren, sondern es geht um die Speicherung großer Informationsmengen, die Selektion darin nach mehr oder minder komplizierten Regeln und die Änderung der Daten. Daher unterscheiden sich kommerzielle Programme von den technisch-wissenschaftlichen, für die FORTRAN entwickelt worden war, typisch durch die folgenden Merkmale:

- Die Arithmetik spielt eine untergeordnete Rolle, das heißt, es wird nicht viel gerechnet. In der Hauptsache handelt es sich um einfache Operationen, nicht um komplizierte Formeln.
- Sehr große Mengen von Daten müssen gespeichert und durchsucht werden.
- Die Präsentation der Ergebnisse, für den Wissenschaftler oft ohne Bedeutung, spielt eine wichtige Rolle, denn mit diesen Ergebnissen, zum Beispiel Lohnlisten, Rechnungen oder Bilanzen, arbeiten Nichtfachleute.
- Von den Programmierern sollten nicht Kenntnisse von Formelsprachen erwartet werden.

1959 bildete sich in den USA, gefördert vom Pentagon, ein Arbeitskreis, der die Entwicklung einer speziell für kommerzielle Programme geeigneten Sprache plante. Er erhielt den Namen CODASYL (Conference on Data Systems Languages). Die Sprache COBOL (Common Business Oriented Language) wurde 1961 erstmals verfügbar [8]. Ihre Normung erfolgte nach zahlreichen Erweiterungen 1969 und 1974 durch das ANSI (American National Standard Institute). COBOL und ALGOL 60 waren damit die ersten Programmiersprachen, die nicht wie FORTRAN von einer einzelnen Firma lanciert, sondern von einer Interessengemeinschaft geschaffen worden waren. Alle folgenden Angaben beziehen sich auf ANSI-COBOL; die Unterschiede zwischen den verschiedenen Dialekten sind zum Teil erheblich; es handelt sich dabei sowohl um weitgehende, teilweise in der Norm vorgesehene Einschränkungen als auch um Erweiterungen gegenüber der Norm.
Der *Stil* der Sprache ist ganz durch die Verwendung von Wortsymbolen geprägt, COBOL-Programme wirken ganz anders als etwa FORTRAN- oder ALGOL-Programme. Dadurch ist COBOL, wie man sagt, «geschwätzig», relativ simple Vorgänge erfordern viel Schreibarbeit. (Spruchweisheit: «COBOL ist lesbar, aber nicht schreibbar»; J. SAMMET: «CO-

BOL is definitely not a succinct language.»)

Der Aufbau von COBOL-Programmen

Ähnlich wie bei FORTRAN ist das *Format* bei COBOL an die klassische Lochkarte gebunden. Jede Zeile ist in die folgenden Abschnitte aufgeteilt:

Bis Spalte 6 kann die Karte numeriert werden.

In Spalte 7 werden Kommentare und ähnliches markiert.

Ab Spalte 8 beginnen die Überschriften. Ab Spalte 12 beginnen alle übrigen Anweisungen.

Das COBOL-System ignoriert die Spalten 73 bis 80; sie können zur Identifikation der Karten benutzt werden. In den nachfolgenden Beispielen sind die Spalten 1 bis 6 und 73 bis 80 weggelassen, sie brauchen ohnehin nicht verwendet zu werden.

Vergleicht man COBOL mit FORTRAN, so fällt zunächst die veränderte Perspektive auf: FORTRAN (in noch höherem Maße ALGOL 60) ist von innen nach außen gebaut, das heißt, im Zentrum des Interesses stehen die an den Daten vorzunehmenden (mathematischen) Operationen. Ein- und Ausgabe sind Hilfsoperationen, ohne die der eigentliche Zweck nicht erreicht werden kann. Bei COBOL ist es gerade umgekehrt: Ein- und (ganz besonders) Ausgabe sind der eigentliche Zweck eines COBOL-Programms, die Verarbeitung ist untergeordnet und wird relativ stiefmütterlich behandelt.

Am deutlichsten wird dieser Unterschied bei den *Zahlen*, die ständig so behandelt werden, wie sie in der Ein- oder Ausgabe stehen, also als Dezimalbrüche, zum Beispiel «398», «-17.04» oder «0.07856». Im Falle von «398» sind also tatsächlich «3», «9» und «8» gespeichert. (Vorzeichen und Punkt werden etwas anders, aber grundsätzlich ähnlich behandelt.) Ein FORTRAN-System würde dagegen (in der Regel) die Zahl bei der Eingabe in die binäre Form verwandeln, also in «110001110», und alle Berechnungen mit dieser dem Rechner bequemeren internen Form durchführen, bis schließlich für die Ergebnisse die Umwandlung (Konvertierung) wieder rückgäng gemacht wird. Will man kaum rechnen, so ist es natürlich vorteilhaft, gleich bei der dezimalen Form zu bleiben. (Da natürlich die Rechner in jedem Fall binär arbeiten, verwendet man hierzu eine spezielle Darstellung, genannt BCD für «binary coded decimal». Dabei werden jeweils pro Ziffer mindestens vier Bits benötigt.) Aus den genannten Gründen gibt es in COBOL auch die Exponentialdarstellung nicht (in FORTRAN zum Beispiel «17.5E-6» für «17,5 mal 10^{-6}»).

In COBOL wurde erstmals eine Unterteilung des Programms eingeführt mit dem Ziel, die wichtigsten Aspekte zu trennen, nämlich die Kennzeichnung des Programms, die Rechnerkonfiguration, auf der es übersetzt und ausgeführt werden kann, die Daten des Programms und schließlich die eigentlichen Programmanweisungen. Ein COBOL-Programm besteht daher aus den vier Teilen

Identification Division
Environment Division
Data Division
Procedure Division

Die *Identification Division* enthält nur, wie der Name sagt, Namen des Programms und seines Autors, also Angaben zur Einordnung und Identifikation des Programms. Diese Angaben müssen in einer festen Reihenfolge erscheinen.

Die *Environment Division* gibt an, auf welcher Maschine das Programm übersetzt und auf welcher es ausgeführt wird. Damit war ein Versuch gemacht, die Maschinenabhängigkeit im Programm selbst zu dokumentieren und dadurch eine Übertragung zu erleichtern.

Eine wesentliche Neuerung stellt die *Data Division* dar, in der komplizierte Daten und Dateien präzise definiert werden können (siehe unten). Die Definition von Files ist auf die übliche Organisation in Blöcke aus fester Anzahl von Records, das heißt traditionell Lochkartenformaten, abgestimmt; hier wurde, wie auch sonst, im Konfliktfall zwischen Allgemeingültigkeit und Einsetzbarkeit ganz pragmatisch entschieden.

Die *Procedure Division* besteht aus einer Hierarchie mit (von oben nach unten) Kapiteln (Sections), Paragraphen, Sätzen und Anweisungen. Sätze sind bis Spalte 12 eingerückt und durch einen Punkt abgeschlossen (Anweisungen durch ein Semikolon, das auch fehlen darf). Die Anfänge der Kapitel (mit Wortsymbol SECTION) und der Paragraphen beginnen in Spalte 8. Es stehen viele Operationen für Aufgaben wie Sortieren, Ausdrucken, Lesen und Schreiben von Dateien zur Verfügung.

Die Konstrukte für die Darstellung von Ablaufstrukturen, also etwa für Schleifen, sind in COBOL geradezu erschütternd unsicher und meist armselig. Ganz besonders schlimm ist das Fehlen echter Prozeduren, der Ersatz (PERFORM) ist sehr unzureichend, weil er nicht mit Parametern ausgestattet werden darf und jede beliebige Folge von Kapiteln (Sections) oder Paragraphen *auch* als Unterprogramm dienen kann.

Ein- und Ausgabe von Daten, Files

Nur kleine Beispiel-Programme rechnen mit fest darin eingesetzten Zahlen, fast alle sinnvollen Programme benötigen Daten als Eingabe von außen. Die Ergebnisse müssen ebenfalls wieder in eine dem Benutzer lesbare Form gebracht oder zur späteren Weiterverwendung auf Magnetbändern oder anderen Speichermedien archiviert werden. Daher spielt die Ein- und Ausgabe bei den Programmiersprachen eine wesentliche Rolle.

FORTRAN hat einige relativ einfache Konstruktionen: Mit READ und WRITE kann auf alle Files zugegriffen werden, zum Beispiel auf Kartenleser, Magnetbandeinheiten oder Schnelldrucker. Die Form der Daten ist in speziellen sogenannten FORMAT-Angaben enthalten; diese werden nicht eigentlich compiliert, sondern erst bei Ausführung der Programme ausgewertet (interpretiert).

In COBOL wurde eine einfache und anschauliche Form gewählt, die Daten darzustellen. Sie besteht darin, ein Schema anzugeben, dem die Daten entsprechen sollen, die sogenannte *Maske*. So steht etwa «99» für eine zweiziffrige Zahl, «AAB9.99» für eine Zeichenfolge aus zwei Buchstaben, einem Leerzeichen (Blank) und einer dreistelligen Zahl mit Dezimalpunkt hinter der ersten Stelle (zum Beispiel «XY 0.38»). Komplizierte Anordnungen können mit weiteren Symbolen aufgebaut werden. Auf diese Weise ist klar, wie die Zahl bei der Ein- oder Ausgabe dargestellt ist.

Für Dateien (Files) sind weitere Angaben notwendig, auf die hier nicht näher eingegangen werden kann.

Datenstrukturen

In FORTRAN lassen sich Felder (Arrays), also Listen oder Matrizen gleichartiger Daten, definieren. In der Praxis ist es oft zweckmäßig, auch verschiedenartige Daten zusammenzufassen. Beispielsweise werden von einem Angestellten Name und Adresse als Zeichenketten (Strings) unterschiedlicher Länge gespeichert, sein Salär ist eine Zahl. Bei Meßwerten müssen außer dem Wert selbst auch Datum und Zeit gespeichert werden.
COBOL löst erstmals dieses Problem, indem es die Möglichkeit bereitstellt, Informationen verschiedenen Typs miteinander zu organisieren. Das nachfolgende Beispiel zeigt die Information von einer Stempelkarte.

```
01  WOCHENARBEITSZEIT
    02  NAME
        03  NACHNAME          PICTURE IS A(20).
        03  VORNAME           PICTURE IS A(15).
        03  TITEL             PICTURE IS A(10).
    02  WOCHE
        03  WOCHEN-NUMMER     PICTURE IS 99.
        03  JAHR              PICTURE IS 9999.
    02  STUNDEN               PICTURE IS 99.99.
```

Man erkennt leicht die Baumstruktur dieser Definition: WOCHENARBEITSZEIT besteht aus NAME und aus WOCHE, WOCHE wiederum aus WOCHEN-NUMMER und JAHR usw. Damit ist es möglich geworden, nach Belieben mit der *ganzen* Information zu arbeiten (wenn beispielsweise die Wochenarbeitszeiten der Vorjahre gelöscht oder anders archiviert werden) oder Teile herauszuziehen, zum Beispiel den Nachnamen oder die Wochenangabe.
In den letzten Jahren hat sich immer mehr die Ansicht durchgesetzt, daß die Datenstrukturen – zusammen mit den darauf möglichen Operationen – den wichtigsten Aspekt einer Programmiersprache darstellen. Darauf wird noch im Zusammenhang mit den abstrakten Datentypen eingegangen werden.

Beispiel

Das folgende Beispiel zeigt wieder die Berechnung der Fakultät einer eingelesenen Zahl. Es ist zu beachten, daß das Beispiel für COBOL untypisch ist, da es weder Files noch aufwendige Datenstrukturen verwendet. Das Beispiel entspricht dem COBOL auf der VAX unter VMS.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FAKULTAET-PROGRAMM.
AUTHOR. J. LUDEWIG BACH DEN BUCH VOM SINGER.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX-11.
OBJECT-COMPUTER. VAX-11.

SPECIAL-NAMES.
    CARD-READER  IS TASTATUR
    LINE-PRINTER IS BILDSCHIRM.

DATA DIVISION.

WORKING-STORAGE SECTION.
77  N              PICTURE IS 99.
77  N-AUS          PICTURE IS Z9.
77  FAKULTAET      PICTURE IS 9(18).
77  FAKULTAET-AUS  PICTURE IS ZZZBZZZBZZZBZZZBZZZBZZ9.

PROCEDURE DIVISION.

FAKULTAET-BERECHNEN SECTION.
ANFANG.
    DISPLAY SPACE "BITTE ZWEISTELLIGE ZAHL "
-                 UPON BILDSCHIRM WITH NO ADVANCING
    ACCEPT N FROM TASTATUR.
    IF N EQUAL TO ZERO GO TO ENDE.
    MOVE N TO N-AUS.
    MOVE 1 TO FAKULTAET.
SCHLEIFE.
    MULTIPLY N BY FAKULTAET
      ON SIZE ERROR GO TO UEBERLAUF.
    SUBTRACT 1 FROM N
    IF N GREATER THAN 1 GO TO SCHLEIFE.
    MOVE FAKULTAET TO FAKULTAET-AUS.
    DISPLAY SPACE N-AUS " HAT FAKULTAET "
-                   FAKULTAET-AUS UPON BILDSCHIRM.
    GO TO ANFANG.

UEBERLAUF.
    DISPLAY SPACE, N-AUS, " ZU GROSS." UPON BILDSCHIRM
    GO TO ANFANG.

ENDE.
    DISPLAY "*** ENDE COBOL-FAKULTAET ***" UPON BILDSCHIRM
    STOP RUN.
```

Weitere Merkmale

COBOL hat eine Syntax, die gelegentlich unsystematisch erscheint und daher das Lernen erschwert. Beispielsweise gibt es die vier Grundrechenarten jeweils mit dem zweiten Operanden als Resultat oder mit separatem Resultat. Bei der Multiplikation lauten die beiden Formen:

```
MULTIPLY A BY B       oder     MULTIPLY A BY B GIVING C
```

Bei der Addition heißt es hingegen:

```
ADD A TO B            oder     ADD A, B GIVING C
```

Für die Division beispielsweise sind in der Grammatik fünf verschiedene Grundformen (jeweils mit vielen verschiedenen Varianten) angegeben.
Die Declaratives erlauben es, Fehlersituationen, zum Beispiel Lesefehler bei der Eingabe, systematisch zu behandeln. In neueren Sprachen entsprechen die sogenannten On-Conditions oder Exceptions diesem Vorbild.
Große COBOL-Systeme besitzen auch einen sprachlich integrierten Preprozessor, der Library-Routinen einbindet und darin Textersetzungen vornimmt, Mechanismen zur Segmentierung von Programmen (falls der Speicherplatz nicht ausreicht), einen Reportgenerator, der wesentliche Funktionen automatisiert (zum Beispiel Seitenaufteilung, Numerierung, Kopf- und Fußzeilen), und andere Hilfssysteme.
Zusammenfassend kann gesagt werden, daß COBOL einige wichtige Neuerungen gebracht hat, vor allem für die Beschreibung der Daten und ihrer Ein- und Ausgabe, sowohl für Dateien wie auch für den Drucker. Der Stil bleibt demjenigen, der an normale, das Formalhafte betonende Programmiersprachen gewöhnt ist, fremd. Die Konstrukte für den Ablauf und das Fehlen echter Unterprogramme erscheinen als Anachronismus. COBOL ist wohl die meistprogrammierte Sprache der Welt und sicher die meistausgeführte, denn es handelt sich ja typisch um Programme bei Banken, Versicherungen und Verwaltungen, die ständig eingesetzt werden.

13 TR

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, erscheint im Frühjahr 1985 im Bibliographischen Institut in Mannheim.

Literatur

1 Sammet J.: Programming Languages: History and Fundamentals. Prentice Hall, 1969.
2 Horowitz E.: Fundamentals of programming languages. Springer-Verlag, Berlin, Heidelberg, New York, 1983.
3 Seegmüller G.: Einführung in die Systemprogrammierung, Reihe Informatik/11. Bibliographisches Institut, Zürich, 1974.
4 Becker A., Haberfellner R., Liebetreu G.: EDV-Wissen für Anwender: ein Handbuch für die Praxis. Verlag Industrielle Organisation, Zürich, 1982.
5 Barron D. W.: Assembler und Lader. Carl Hanser Verlag, München, 1970.
6 McCracken D. D.: FORTRAN in der technischen Anwendung. Carl Hanser Verlag, München, 2. Aufl., 1977.
7 Spiess W. E., Rheingans G.: Einführung in das Programmieren mit FORTRAN, De Gruyter Verlag, Berlin, 1980.
8 Singer F.: Programmierung mit COBOL, Teubner-Studienskripten. B. G. Teubner, Stuttgart, 5. Aufl., 1983.