Die Wahl einer geeigneten Programmiersprache gilt mit Recht als wichtige Entscheidung, die erhebliche Auswirkungen auf die Leistung der Programmierer und auf die Qualität ihrer Produkte hat.

Entwurf und Auswahl von Programmiersprachen

Von Jochen Ludewig und Helmut Sandmayr

Dieser Artikel legt nach einer Abgrenzung des Themas zunächst dar, welche Einflüsse eine Programmiersprache formen und welche Merkmale sich bis heute allgemein herausgebildet haben; die wichtigsten werden näher betrachtet. Es folgen Hinweise auf Kriterien, anhand derer Programmiersprachen beurteilt werden können. Am Schluß steht ein Ausblick auf die zukünftige Entwicklung.

Programmiersprachen — ein beliebtes Thema

Fachleute wie auch Laien der Informatik diskutieren gerne über Programmiersprachen: Jeder hat gewisse Kenntnisse, Erfahrungen, Zu- und Abneigungen. Dabei hört man vielfach Aussagen wie:
«ADA hat sehr gute Fehlermeldungen.»
«FORTRAN ergibt die kürzeste Ausführungszeit.»
«PASCAL ist mir zu kompliziert.»
Hier ist offenbar nicht nur von der Sprache, sondern von etwas Umfassendem, dem Programmiersystem, die Rede. Die Qualität der Fehlermeldungen ist bestimmt durch den Übersetzer oder Interpreter; dies gilt auch für die Ausführungszeit, wobei natürlich schließlich die Hardware der entscheidende Faktor ist; die Kompliziertheit einer Sprache ist häufig nur durch schlechte Unterlagen (Handbücher) suggeriert (oder einfach vermutet). Wir wollen also zunächst eine begriffliche Trennung vornehmen: Die *Sprache* ist eine rein abstrakte Sache, bestehend aus einer Menge von *Regeln*, was in der Sprache zulässig und damit korrekt ist, und einer *Erklärung*, was es (falls es korrekt ist) bedeutet. Die Regeln nennen wir *Syntax*, die Erklärung dazu *Semantik*.
Die Syntax sagt also beispielsweise aus, daß das folgende ein korrektes Programm der Sprache PASCAL ist:

```
PROGRAM beispiel (input, output);
BEGIN
writeln ('Das war schon alles.')
END.
```

Die Semantik sagt dazu aus, daß eine Ausführung des Programms die einmalige Ausgabe des Satzes «Das war schon alles.» auf einem (durch die Sprache nicht definierten) Gerät, zum Beispiel auf dem Drucker, zur Folge hat.
Vielfach versucht man, die Syntax wenigstens teilweise *formal* zu definieren (am häufigsten nach dem Schema der BNF oder BACKUS-NAUR-Form) und dreht dann die Definition gerade um: Syntax ist, was sich formal definieren läßt. So ist es etwa in der Beschreibung der Sprache ALGOL 60 geschehen. Die zu Beginn gewählte Definition ist aber letztlich die klarere.
Zum *Programmiersystem* gehören alle Werkzeuge, die die Sprache praktisch verwendbar machen (wenn man von der Verwendung zum Gedankenaustausch zwischen Menschen absieht, die bei ALGOL 60 zu den Zielsetzungen zählte), also

- der *Übersetzer* (Compiler), der unser Programm in eine von der Maschine ausführbare Form transformiert
- das *Laufzeitsystem*, das während der Programmausführung Hilfestellung leistet bei komplizierten Operationen, zum Beispiel bei der Behandlung einer Division durch Null, und dabei seinerseits auf das Betriebssystem zugreift
- *Programmbibliotheken*, die uns häufig gebrauchte Programmteile zur Verfügung stellen, etwa für trigonometrische Funktionen oder für die Matrizenrechnung
- eventuell noch *Hilfssoftware*, die uns Arbeiten an den Programmen speziell dieser Sprache erleichtern, zum Beispiel Formatierprogramme oder Testsysteme

Bei gewissen Sprachen (zum Beispiel BASIC) besteht die Möglichkeit (aber keineswegs die Notwendigkeit), fast alle Aufgaben des Übersetzers in das Laufzeitsystem zu verlagern; man bezeichnet dieses dann als *Interpreter*.
Offenbar ist die Qualität eines Programmiersystems nur zu einem kleinen Teil von der Programmiersprache, vorwiegend aber von den zugehörigen Werkzeugen bestimmt, die meist von derselben Firma wie die Rechnerhardware stammen. Das Programmiersystem muß rechtzeitig und zu einem vertretbaren Preis verfügbar sein, es muß ausreichend effizient, korrekt und zuverlässig

Geschichte

Ansätze zu Sprachen, in denen Rechenabläufe formal beschrieben werden können, hat es schon im 19. Jahrhundert gegeben; Countess AUGUSTA ADA LOVELACE, die Tochter Lord BYRONS, wurde darum jüngst zur Patronin der Sprache ADA gewählt. Aber praktische Bedeutung erreichten ihre Ansätze nie. Erst mit der Konstruktion eines Rechners, dessen Steuerung nicht mehr durch Schalter oder durch einen Lochstreifen erfolgt, sondern durch Befehle, die wie Daten gespeichert sind und (prinzipiell) einer nach dem anderen ausgeführt werden, war der Weg frei für Programmiersprachen. Eine solche Maschine, wurde etwa 1946 in den USA von ECKERT und MAUCHLY konzipiert, wir bezeichnen sie heute nach dem Miterfinder, der die Ideen zu Papier gebracht hat, als VON-NEUMANN-*Rechner*.
Sehr bald zeigte sich, wie aufwendig es ist, alle elementaren Operationen im Detail und in der binären Sprache des Computers zu formulieren. Die Programmierer benutzten also das Werkzeug, dessen ursprünglicher Zweck es war, technische oder physikalische Berechnungen zu erleichtern, zur Vereinfachung ihrer eigenen Arbeit, sie programmierten Übersetzer. Zunächst waren dies Assembler, also einfache Programme, die einen mnemonischen Code in den Maschinencode übertrugen. Bald kam die automatische Zuordnung der Speicheradressen hinzu, die von Hand äußerst mühsam und fehlerträchtig ist. Den eigentlichen Durchbruch aber brachte – schon ein knappes Jahrzehnt nach

Dr. rer. nat. JOCHEN LUDEWIG ist Leiter des Projekts Software Engineering am Brown-Boveri-Forschungszentrum in Dättwil. Dr. sc. math. ETH HELMUT SANDMAYR ist Leiter der Software-Entwicklung Netzleittechnik bei BBC in Baden.

arbeiten, der Compiler muß brauchbare Fehlermeldungen liefern usw. Außer in Fällen, wo die Sprache einen entscheidenden Einfluß auf das Programmiersystem hat, sind zu diesem Punkt keine allgemeinen, von der jeweiligen Implementierung unabhängigen Aussagen möglich. Wir müssen uns hier also auf die Sprache selbst beschränken.

Eine weitere Abgrenzung ist notwendig. Wir behandeln vor allem konventionelle Sprachen für technisch-wissenschaftliche Anwendungen; weder wollen wir die Probleme der kommerziellen Programmierung, die ganz vornehmlich in der Uralt-Sprache COBOL erfolgt, einbeziehen, noch die neuesten Entwicklungen in Richtung auf *funktionale* und *logische* Sprachen (mit Einschränkungen repräsentiert durch LISP und PROLOG) diskutieren. «Konventionell» heißt hier, daß wir vom Prinzip der Von-Neumann-Maschine ausgehen (siehe Kasten «Geschichte der Programmiersprachen»).

Einflüsse auf Programmiersprachen

Der ursprüngliche Zweck einer Programmiersprache ist es, Problemlösungen so formulierbar zu machen, daß sie von einem Rechner ausgeführt werden können. Daraus ergeben sich zwei dominierende Einflüsse: die Architektur der Zielmaschine (Rechner und Betriebssystem) und das Anwendungsgebiet. Ebenfalls wichtig sind der Stand der Implementationstechnik und des Software-Engineerings. Hinzu kommen beim Entwurf einer Sprache noch spezielle Ziele wie Einfachheit, Orthogonalität (für jeden Zweck sollte es nach Möglichkeit genau *ein* Sprachkonstrukt geben) oder Unterstützung der Programmentwicklung im Rahmen großer Projekte.

Rechnerarchitektur, Betriebssysteme

Alle heute in größerem Stil eingesetzten Sprachen sind geprägt durch die Rechnerarchitektur nach JOHN VON NEUMANN. In der etwa 35jährigen Geschichte dieser Architektur wurde die

der Programmiersprachen

dem Von-Neumann-Rechner – die Sprache FORTRAN. Der Name entstand als Abkürzung von Formula Translation, denn der wichtigste Fortschritt war, daß nun die Formeln nicht mehr in elementare Rechenoperationen zerlegt werden mußten, sondern direkt ins Programm gesetzt werden konnten.

FORTRAN war ganz pragmatisch realisiert worden, und fast im gleichen Stil wurde es noch vor wenigen Jahren verbessert (FORTRAN 77). Schon Ende der fünfziger Jahre aber gab es Ansätze, eine Programmiersprache systematisch zu entwickeln. Das Ergebnis war ALGOL 60 (Algorithmic Language 1960), wie FORTRAN ein großer Wurf, aber von ganz anderem Charakter. Definiert von einer Kommission meist europäischer Wissenschaftler, war ALGOL 60 ein wesentlicher Fortschritt gegenüber den meisten seiner *Nachfolger* (DAVID GRIESS). Wesentliche Prinzipien wie Rekursion und Deklaration wurden hier erstmals praktisch angewendet. Die Sprache setzte sich zwar in den USA nie durch, hatte aber durch die europäischen Universitäten wesentlichen Einfluß auf die Entwicklung.

Mit dem Ziel, die rasch wachsenden Möglichkeiten der Rechner voll nutzbar zu machen, entstand ab 1964 bei IBM die Sprache PL/I (die Abkürzung steht für das bescheidene «Programming Language I»). Darin wurden Konzepte aller gängigen Sprachen, also FORTRAN, ALGOL 60 und COBOL, gemischt, PL/I sollte die Universalsprache sein. So entstand ein Saurier der Informatik, mächtig und gefürchtet. Ähnlich wie ALGOL 60 eine Antwort auf FORTRAN war, entwikkelte man in Europa nun ALGOL 68. Um im Bild zu bleiben, ein etwas eleganterer Saurier, dem allerdings die Unterstützung einer starken Mutter fehlte und der daher nie große Bedeutung erlangt hat.

Die sechziger Jahre waren die Dekade der Programmiersprachen. Einerseits gab es noch wenig gesichertes Wissen, so daß viel zu erkunden war, anderseits waren die Techniken und Hilfsmittel weit genug gediehen, um Versuche mit neuen Sprachen zu ermöglichen. So gehörte es beispielsweise für Doktoranden auf dem gerade entstehenden Gebiet der Informatik geradezu zum guten Ton, eine eigene Sprache zu definieren und den Nachweis ihrer besonderen Eignung zu versuchen. Das ausgezeichnete Buch von SAMMET (1969) markiert den Höhepunkt dieser Entwicklung.

Sehr bald verlor die Glitzerfassade der allmächtigen Computer an Glanz, das Schlagwort von der Softwarekrise ging um. Damit tauchten neue Ziele auf: Eine Programmiersprache soll dazu beitragen, daß Programme möglichst wenig Fehler enthalten, daß sie leicht verständlich und problemlos zu ändern sind. PASCAL, eine betont kleine Sprache in der Tradition von ALGOL 60, aber durch neue Elemente vor allem zur Definition von Datenstrukturen auf dem Stand von 1968, eroberte Europa und war einige Jahre später auch in USA erfolgreich.

In den siebziger Jahren ist es um dieses Thema ruhiger geworden. Es entstanden vor allem noch zwei Sprachen, etwa gleichzeitig und mit ähnlicher Zielsetzung: MODULA-2 als logische Fortsetzung von PASCAL, ADA als Standardsprache des US-Verteidigungsministeriums, das mit seinem astronomischen Softwarebudget den Trend bestimmen kann. MODULA-2 ist handlich und speziell für Personal Computers gut geeignet, ADA komfortabel und mit allen Extras ausgestattet, die für die Realisierung sehr großer Systeme erforderlich sind (dazu gehört auch eine ungewöhnlich strenge Standardisierung). Beide Sprachen repräsentieren den heutigen Stand der Technik, vor allem im Hinblick auf die Notwendigkeit, große Programme aus Bausteinen zusammenzusetzen, ohne an den Schnittstellen auf die Konsistenzprüfungen verzichten zu müssen.

Viele andere Sprachen können den oben erwähnten als Vorstufen oder Weiterentwicklungen, Erweiterungen oder Teilsprachen zugeordnet werden. Außerdem gibt es aber auch Nebenlinien der Sprachenevolution, die schon seit langem von der Hauptlinie getrennt laufen. Der bekannteste Sonderfall dieser Art ist BASIC (Beginner's All Purpose Symbolic Instruction Code), entstanden 1965 in den Vereinigten Staaten, dem Namen entsprechend als Anfängersprache. Vor allem die Welle der Personalcomputer hat BASIC nach oben gespült, trotz des entschiedenen Widerstands vieler Fachleute, die dieser Sprache vorwerfen, gerade zu verhindern, was nach dem Stand der Programmiertechnik erforderlich ist (beispielsweise klare Ablaufstrukturen und aussagekräftige Namen).

Arbeitsgeschwindigkeit der Rechner um einige Größenordnungen erhöht und der verfügbare Speicherplatz im gleichen Maße vergrößert. Dies erlaubt es uns heute, auf die volle Ausnutzung der technischen Möglichkeiten, wie sie die Assemblerprogrammierung prinzipiell zuläßt, zu verzichten. Anderseits zwingt uns der – jetzt technisch erfüllbare – Wunsch nach Programmen für immer komplexere Probleme dazu, höhere Sprachen zu verwenden, die unseren Kopf von vielen unwesentlichen Details entlasten.

Auch sonst ist die Wechselwirkung zwischen Sprachen und Architektur der Zielmaschinen sehr stark: Vorhandene Architekturmerkmale beeinflussen den Entwurf von Sprachen, umgekehrt führen attraktive Sprachkonzepte, die sich auf herkömmlichen Rechnern nicht effizient implementieren lassen, zu neuen Ansätzen im Bereich der Architektur. Da Konzepte aber oft in verschiedenen Bereichen gleichzeitig reifen, ist es gelegentlich nicht möglich zu entscheiden, ob der Anstoß von der Sprachen- oder von der Architekturseite gekommen war. Bei den folgenden zwei Beispielen ist jedoch die Reihenfolge offenkundig.

- Die bereits in FORTRAN und ALGOL 60 vorhandenen Parametermechanismen zum Informationsaustausch zwischen Programm und Unterprogramm sind inzwischen in verschiedenen Rechnern durch spezielle Befehle realisiert.
- Gleichzeitig arbeitende Mikroprozessoren erreichen bei geeigneter Verteilung der Aufgabe eine einem einzelnen Großrechner vergleichbare Leistung. Die Programmierung einer solchen Hardware ist aber mit konventionellen Sprachen für Von-Neumann-Rechner kaum möglich. Ein alternativer Ansatz ist das Datenflußkonzept, bei dem die Daten Ströme zwischen den Operatoren bilden. Eine Operation wird ausgeführt, wenn die benötigten Daten bereitstehen. Eine geeignete Programmiersprache für die Formulierung von Programmen solcher *Datenflußrechner* ist noch immer eine Herausforderung für die Sprachdesigner.

Anwendungsgebiet

Es ist offensichtlich von Vorteil, wenn der Anwender seine Lösungen in der ihm geläufigen Terminologie oder einer verwandten Sprache formulieren kann. So gab es früh die Unterscheidung zwischen Sprachen für technisch-wissenschaftliche Probleme mit dem Schwerpunkt auf der Auswertung von Formeln und Funktionen (FORTRAN, ALGOL 60) und Sprachen für kommerzielle Anwendungen mit nichtnumerischen Elementen, also Texten und anderen Strukturen zur Aufnahme von Informationen, wie sie auch auf Karteikarten geführt wurden (COBOL). Weitere Spezialisierung lieferte Sprachen für die Prozeßautomatisierung (PEARL), für die Simulation (SIMULA), für Anwendungen in der Mathematik, der Logik und der künstlichen Intelligenz (PROLOG, LISP), für einmalige Berechnungen (die sogenannten Wegwerfprogramme, BASIC) sowie Sprachen für Kinder (LOGO) und andere Zwecke.

Software-Engineering

Bestimmte Methoden der Softwareentwicklung, wie man sie unter dem Sammelbegriff «Software-Engineering» zusammenfaßt, können durch die Programmiersprache unterstützt oder auch sabotiert werden. Eine Sprache (wie zum Beispiel BASIC), in der sich ein Programm nicht in Einheiten mit einfachen Schnittstellen zerlegen läßt, eignet sich nicht für ein Projekt, an dem verschiedene Personen gleichzeitig entwickeln sollen. Große Programme oder Programmsysteme müssen sich aus Komponenten aufbauen lassen, die jeweils weitgehend abgeschlossene Einheiten bilden und als solche auch von den Werkzeugen wie Compilern oder Verwaltungssystemen bearbeitet werden können. Dabei ist es auch von großer Bedeutung, daß die Werkzeuge prüfen können, ob die Schnittstellen von beiden Seiten konsistent realisiert sind (wie beispielsweise in ADA).

Andere Entwurfsziele

Neben den drei bereits genannten Aspekten spielen beim Entwurf (oder bei der Auswahl) einer Programmiersprache noch eine Reihe anderer Ziele eine Rolle, beispielsweise einfache Implementierbarkeit (PASCAL), Effizienz der Ausführung (mit geringem Aufwand erreichbar in FORTRAN), leichte Erlernbarkeit für Gelegenheitsbenutzer (BASIC) oder Orthogonalität der Sprachelemente (ALGOL 68).

Einige wichtige Merkmale und Elemente von Programmiersprachen

Formal ist eine Programmiersprache gekennzeichnet durch die Menge der darin verfügbaren Objekte und durch die Operationen, die mit diesen Objekten ausgeführt werden können. Bei ganz primitiven Sprachen kann die Beschreibung tatsächlich auf dieser Ebene erfolgen (ein einfacher Taschenrechner wird beispielsweise gesteuert durch die Eingabe von Zahlen und Rechenoperationen); bei echten Programmiersprachen sind noch Konstrukte zur Gliederung notwendig, sowohl für die Daten (Typkonzept) als auch für die Operationen (Ablaufstrukturen) und für die Organisation des Programms insgesamt (zum Beispiel Modul- oder Prozedurkonzept). Der Charakter der Sprache ist geprägt durch das Vorhandensein oder Fehlen gewisser Elemente dieser Kategorien. Nachfolgend wird versucht, die wichtigsten Merkmale vorzustellen.

Ausdrücke und Wertzuweisungen

Formeln, wie man sie beispielsweise in Formelsammlungen für Ingenieure findet, werden in Programmen dargestellt durch Ausdrücke und Wertzuweisungen, zum Beispiel

```
y := a* x**2+b*x+c
```

Der Ausdruck rechts vom Symbol «:=» bestimmt den Wert, der der Variablen auf der linken Seite zugewiesen wird. In FORTRAN und anderen älteren Sprachen verwendet man für die Wertzuweisung einfach das Gleichheitszeichen, aber das ist mathematisch inkorrekt, denn die Bedeutung ist ja nicht «*ist* gleich», sondern «*setze* gleich».

Welche Operationen die Sprache zur Bildung von Ausdrücken anbietet, ist bestimmt durch die darin vorkommenden Typen.

Typen

Der klassische Von-Neumann-Rechner arbeitet nur mit den Inhalten seiner Speicherzellen; ob diese Inhalte ganze Zahlen, Wahrheitswerte, einzelne Buchstaben oder Fragmente aus gespeicherten Texten bedeuten, ist dabei ohne Belang. Natürlich ist es aber in der Regel nicht sinnvoll, Buchstaben zu addieren, aus Wahrheitswerten die Wurzel zu ziehen oder Texte zu multiplizieren; Operationen sind jeweils nur für Operanden gewisser Typen anwendbar. Bei den maschinennahen Sprachen (Assemblersprachen) muß der Benutzer selbst darauf achten, daß in dieser Hinsicht nichts durcheinandergerät. Höhere Sprachen helfen ihm dabei durch *Typen*, beispielsweise die Typen INTEGER, REAL und LOGICAL in FORTRAN. Ein Typ ist gekennzeichnet durch die Menge der möglichen Werte (beispielsweise TRUE und FALSE für LOGICAL) und durch die Operationen, die auf Objekte dieses Typs zulässig sind. Mit der Information, daß eine Variable

einem bestimmten Typ zugeordnet ist, kann der Übersetzer dreierlei anfangen:

- Erkennung unzulässiger Operationen (Addition von LOGICAL)
- Auswahl der richtigen Operation («+» bedeutet ganzzahlige Addition, wenn die Operanden vom Typ INTEGER sind, aber Gleitkommaaddition für REAL-Operanden)
- automatische Anpassung der Operanden (bei Addition einer INTEGER-Zahl und einer REAL-Zahl wird erstere automatisch in eine REAL-Zahl verwandelt).

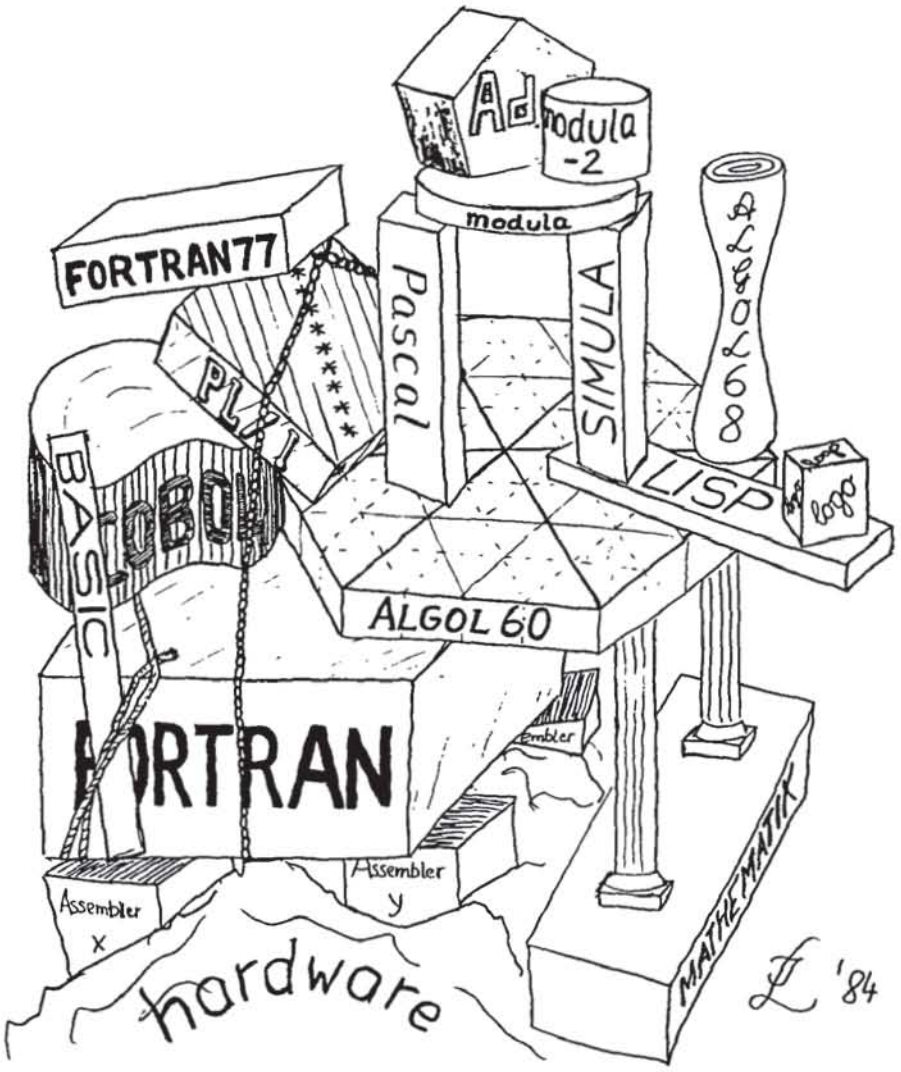
Früher hatten die beiden letzten Möglichkeiten im Vordergrund gestanden, also die Erhöhung des Komforts. Als sich zeigte, daß Programmfehler ein zentrales Problem darstellen, gewann die erste Möglichkeit an Bedeutung. Man ging daher noch einen Schritt weiter und verlangte, daß der Programmierer (anders als in FORTRAN) jedes Objekt explizit einführen müsse. Diese *vollständige Deklaration* gestattet nicht nur die Prüfung, ob das Objekt nur entsprechend seinem Typ verwendet wird, sondern hilft auch ganz wesentlich, Schreibfehler zu entdecken. (Schreibt man in FORTRAN anstelle der Variablen KONV versehentlich CONV, so hat man unabsichtlich eine neue Variable eingeführt, die an dieser Stelle statt KONV verwendet wird. Der Compiler merkt nichts und meldet natürlich auch nichts.)

Programme arbeiten meist nicht mit elementaren Objekten wie einzelnen Zahlen, sondern mit zusammengesetzten Daten, beispielsweise Personendaten für eine ganze Gemeinde. Hierfür sehen die Programmiersprachen *Datenstrukturen* vor. Die wichtigsten sind:

- Felder (Arrays) aus gleichartigen Elementen
- Verbunde (Records) aus im allgemeinen verschiedenartigen Elementen
- Mengen (Sets) aus gleichartigen Elementen, zum Beispiel verschiedenen Buchstaben
- Netzwerke mit Zeigern (Pointers)

FORTRAN und ALGOL bieten nur Felder; erst in den sechziger Jahren erkannte man die volle Bedeutung der Datenstrukturen. Daher kamen mit PASCAL die Verbunde und Netzwerke hinzu. Da es möglich ist, eine Datenstruktur wieder als Baustein einer komplexeren Datenstruktur zu verwenden, steckt darin eine kaum zu überschätzende Bereicherung der Sprache.

Schon oben wurde gesagt, daß der Typ durch die Menge der zulässigen Operationen gekennzeichnet ist. Der Übersetzer kann aber nur schematische Prüfungen vornehmen. So sei uns beispielsweise von einer ganzzahligen Variablen bekannt, daß sie nur um eins erhöht oder um eins gesenkt werden darf. Der Übersetzer kann aber nicht verhindern, daß sie multipliziert oder um 327 erhöht wird. Folglich realisieren wir für die beiden zulässigen Operationen zwei Prozeduren und verbieten jeglichen *direkten* Zugriff auf unsere Zahl; die Prozeduren dürfen aufgerufen werden. Eine solche Gruppe von Prozeduren, die auf Objekte eines bestimmten Typs exklusives Zugriffsrecht haben, bezeichnet man als *abstrakten Datentyp*. Vor allem bei Datenstrukturen in komplexen Programmen ist diese Möglichkeit des «information hiding» (David Parnas) eine ganz wesentliche Erleichterung, und zwar nicht nur bei der Programmentwicklung, sondern mehr noch bei einer späteren Korrektur oder Modifikation.



Viele Baumeister verderben den Stil. *Im Buch von J. Sammet (1969) ist ein babylonischer Turm abgebildet, dessen Steine Programmiersprachen sind. Wie sinnlos ein solcher Turm auch sein mag, so ist er doch auf ein wohldefiniertes Ziel gerichtet. Damit beschönigt der Vergleich aus heutiger Sicht die Situation bei den Programmiersprachen. Tatsächlich stellt sich die Lage eher als Baustelle dar, bei der jeder ein anderes Ziel (oder gar keins) verfolgt und niemand den Überblick hat (Grafik: J. Ludewig).*

Die Idee der abstrakten Datentypen wurde (als Class-Konzept) 1967 mit SIMULA geboren, doch erst MODULA-2 und ADA machen vollen Gebrauch davon. Fachleute betrachten abstrakte Datentypen als die Errungenschaft des Software-Engineerings in den letzten Jahren.

Ablaufstrukturen

Assemblersprachen enthalten spezielle Befehle, mit denen der Programmierer veranlassen kann, daß nicht einfach nach jedem Befehl der nächste ausgeführt wird: die Sprunganweisung («BRANCH», «GOTO») in unbedingter und bedingter Form. Prinzipiell ist der bedingte Sprung allein ausreichend, um alle Algorithmen zu realisieren. Die Erfahrungen haben aber gezeigt, daß Programme durch Sprunganweisungen

sehr unübersichtlich werden («Like a bowl of spaghetti»). Viele Fehler sind die unvermeidliche Folge. Daher bemüht man sich heute, nur solche Ablaufkonstrukte zu verwenden, die *lokal* verständlich sind, also ohne daß man überblickt, was an anderer Stelle geschieht. Zu diesen *abgeschlossenen Konstrukten* gehören neben der einfachen Befehlssequenz

- Schleifen (Laufschleifen, WHILE-Schleifen)
- Verzweigungen (IF-THEN-ELSE, CASE-Anweisung)
- und vor allem Prozeduren.

Der Sprung ist also (fast) verschwunden. Man sieht hier ein wesentliches Merkmal der höheren, «problemorientierten» Sprachen: Obwohl die Maschine nur den Sprung kennt, werden dem Benutzer ganz andere Konstrukte zur Verfügung gestellt, weil sich gezeigt hat, daß damit die Ziele (vor allem Korrektheit) besser erreicht werden.

Programmgliederung

Wenn Programme relativ klein sind, geschieht die Gliederung vor allem, damit wir uns auf abgeschlossene Teile konzentrieren können. Dazu ist vor allem die *Prozedur* geeignet. Sie erlaubt es, für ein abgeschlossenes Teilproblem eine ebenso abgeschlossene Teillösung anzufertigen, die im Kontext nur als Prozeduraufruf, also als Elementaroperation erscheint. Die irrtümliche Verwendung fremder Variablen usw. ist durch die Kontrollmechanismen des Übersetzers weitgehend ausgeschlossen. Indem die Prozedur nur über Parameter mit ihrer Umgebung kommuniziert, kann sie mehrfach mit unterschiedlichen Parametern aufgerufen werden. Dabei ist es wichtig, daß die Schnittstelle vom Übersetzer auf Typkonsistenz der Parameter geprüft wird (ist in FORTRAN nicht der Fall); die Sicherheit gegen Fehler wird auch wesentlich dadurch verbessert, daß für jeden Parameter angegeben werden muß, in welche Richtung er die Information transportiert (in ADA IN, OUT und INOUT).
Große Systeme werden von mehreren Gruppen entwickelt, die ihre Komponenten möglichst unabhängig voneinander entwickeln und testen wollen. Auch ist es bei großem Umfang nicht mehr praktikabel, nach jeder Änderung das gesamte Programm neu zu übersetzen. Aus diesen Bedürfnissen folgt ein Modulkonzept, das sich (wie in MODULA-2 und ADA geschehen) auf sehr natürliche Art mit dem oben skizzierten Konzept der abstrakten Datentypen verbinden läßt.

Ausführungsweise von Programmteilen

Prozeduren (in FORTRAN SUBROUTINES) werden sequentiell ausgeführt, das aufrufende und damit übergeordnete Programm unterbricht seinen Ablauf für die Zeit der Prozedurausführung. In gewissen Fällen hat es Vorteile, wenn sich beide beteiligten Programmteile so verhalten, als seien sie übergeordnet, also jeweils zeitweise auf das andere warten. Man bezeichnet solche gleichgeordnete Programmteile als *Coroutinen;* in MODULA-2 sind sie nicht vorgegeben, aber realisierbar.
Können mehrere Programmteile gleichzeitig (oder «quasi-simultan», also *wie* gleichzeitig) ausgeführt werden, so nennt man sie *Tasks*. Alle Sprachen, die für die Programmierung von Echtzeitaufgaben vorgesehen sind, enthalten ein Task-Konzept, beispielsweise PEARL oder ADA.
Im Gegensatz zu allen bisher genannten Einheiten werden *Exceptions* in ADA (oder ON-CONDITIONS in PL/I) nicht durch Aufruf oder ähnlich gestartet, sondern dadurch, daß ein gewisses Ereignis eintritt, beispielsweise eine Division durch Null.

Aspekte bei der Beurteilung von Programmiersprachen

Wo immer eine Programmiersprache eingesetzt werden soll, kann sich die Beurteilung nicht auf ihre Eigenschaften beschränken, sondern muß die Bedingungen der speziellen Situation einbeziehen, vor allem die Anwendung, die Vorbildung der Mitarbeiter und die vorhandene Infrastruktur mit Hilfsmitteln zur Unterstützung der Programmentwicklung. Da diese Faktoren nicht exakt bewertbar sind, bleibt viel Raum für nichtrationale Einflüsse wie frühere Erfahrungen, Einfluß und Ansehen der für eine Sprache werbenden (manchmal missionierenden) Person oder andere, oft hochgespielte persönliche oder firmenpolitische Gründe. Obwohl solche Einflüsse oft den Ausschlag geben, sollen sie hier nicht weiter analysiert werden; es ist eine allgemeine Aufgabe des Managements, eine technische Frage stufengerecht klären und entscheiden zu lassen.
Statt dessen wollen wir versuchen, uns am allgemeinen Zweck des Einsatzes von Programmiersprachen zu orientieren, der *effizienten Erstellung* eines *korrekten Programms* und dem *Erhalt der Korrektheit* auch bei Änderungen während seiner Benutzung. Eine Programmiersprache ist also danach zu bewerten, inwieweit sie hilft, Fehler zu vermeiden oder wenigstens zu erkennen. Folgende Eigenschaften tragen sicher dazu bei:
Eine ausreichende Mächtigkeit der Sprache erlaubt dem Programmierer, seine Absicht ohne Umformung oder Verschlüsselungen zu formulieren. (Er sagt, was er sagen, nicht, was die Maschine hören will.) Solche Transformationen führt ein Compiler – im Gegensatz zum Menschen – konsistent und zuverlässig aus.
Wenige in der ganzen Sprache gültige Konzepte erleichtern das Erlernen und Beherrschen der Sprache, die wichtigste Voraussetzung für das erfolgreiche Programmieren. So kann sich der Programmierer zum Beispiel leichter merken, daß an allen Stellen eines Programms, an denen ein Wert erwartet wird, ein Ausdruck stehen darf, als sich eine Liste von Ausnahmen einzuprägen, die aus seiner Sicht unverständlich sind (zum Beispiel «Im Befehl A darf ein Wert nur durch eine Konstante dargestellt werden, im Befehl B durch einen Ausdruck mit höchstens einer Multiplikation und einer Addition, im Befehl C ein allgemeiner Ausdruck, usw.»). Diese Vielzahl von Fällen verunsichert den Programmierer und begünstigt einen Programmierstil, der die Möglichkeiten einer Sprache nicht nutzt, sondern nur die einfachsten Varianten verwendet.
Programme werden im allgemeinen nicht nur geschrieben, sondern während ihrer Entwicklung und Benutzung immer wieder verändert. Sie werden also häufig gelesen, und es ist daher wesentlich, daß die Sprache allgemeinen Regeln, beispielsweise der Mathematik,

Beim Entwurf oder bei der Auswahl von Programmiersprachen spielen verschiedene Ziele eine Rolle. Die leichte Erlernbarkeit der Sprache ist nur ein Aspekt (Foto: Chr. Sonderegger, Informatikschule Schweiz).

entspricht; andernfalls kommt es leicht zu Mißverständnissen. So bedeutet zum Beispiel in PL/I

A**2

für eine quadratische Matrix A nicht wie üblich das Produkt A*A, bei dem jedes Element als Skalarprodukt zweier Vektoren entsteht, sondern die Matrix, die man erhält, wenn man jede Komponente der Matrix A *einzeln* quadriert.
Wie bereits erwähnt, sind neben den Eigenschaften der Sprache auch andere Aspekte von Bedeutung, zum Beispiel ihre Standardisierung, ihr Verwendungsgrad und die Verfügbarkeit geeigneter Implementierungen (Übersetzer, Einbettung in das Betriebssystem eines Rechners). Standardisierung ist natürlich kein Wert an sich, das Fehlen eines Standards erschwert aber die Anwendung einer Sprache (es fehlen zuverlässige Lehrbücher und sichere Programmierer) und den später vielleicht erforderlichen Wechsel auf eine andere Maschine. Ein höherer Verwendungsgrad bedeutet im allgemeinen, daß es viele ausgebildete Programmierer gibt und ein Projekt nicht zuerst mit einem Sprachenkurs beginnen muß. (Ein Projekt kann trotzdem mit einem Kurs beginnen, es gibt genügend andere wichtige Themen.) Die Implementation schließlich bestimmt, ob geeignete Werkzeuge für eine Sprache existieren, ob die Sprache effizient einsetzbar ist oder ob man auf eine eigentlich attraktive Sprache verzichten muß, weil es an den Werkzeugen hapert.

Entwicklungstendenzen

Die Ideen und Ziele moderner Sprachen (MODULA-2, ADA) lassen sich wie folgt zusammenfassen:

- Rechner dienen nicht mehr vorrangig zum Rechnen, sondern zur Lösung organisatorischer Aufgaben, zum Suchen, Kombinieren, Umformen von Informationen aller Art. Die Sprachen müssen also mit den dazu notwendigen Basistypen und Operationen ausgestattet sein.
- Zur Beherrschung der Komplexität, die die Programme aufweisen, muß die Sprache starke Hilfe leisten (vor allem durch Modulkonzept, abstrakte Datentypen und strenge Typbindung).
- Präzise Definition und damit Standardisierung der Sprache sind für Austausch und Kombination von Programmen essentiell.

Mit einem Softwarebudget von rund 3 Mia Dollar pro Jahr spielt das U.S. Department of Defense («Pentagon») eine gewichtige Rolle in der Welt der Programmiersprachen. Seine Entscheidung zugunsten von ADA als Standardsprache, die ganz deutlich der europäischen Tradition entstammt (ALGOL, SIMULA, PASCAL), nicht der amerikanischen (FORTRAN, COBOL, PL/I), deutet darauf hin, daß der Trend in Richtung auf Sprachen geht, deren Mächtigkeit und Sicherheit in einem ausgewogenen Verhältnis stehen. [13] TR

Aus Raumgründen ist hier keine Literatur zu einzelnen Programmiersprachen angegeben. Diese kann den folgenden Schriften entnommen werden.
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.
Horowitz E. (1983): Fundamentals of programming languages. Springer-Verlag, Berlin, Heidelberg, New York.
Ludewig J. (1984): Sprachen für die Programmierung – eine Übersicht. Brown-Boveri-Forschungszentrum, KLR-Bericht 84-35 C. Diese Schrift ist als interner Bericht nicht allgemein zugänglich. Er wird aber – voraussichtlich ab November 1984 – in drei Teilen in der «Technischen Rundschau» abgedruckt. Eine erweiterte Fassung erscheint 1985 in der Reihe «Informatik» des Bibliografischen Instituts Mannheim.
McKeeman W. M. (1974): Programming language design. Advanced Course on Compiler Construction. Springer-Verlag.
Sammet J. (1969): Programming Languages: History and Fundamentals. Prentice Hall.