

DER MATHEMATISCHE UND NATURWISSENSCHAFTLICHE UNTERRICHT

1984 , Heft 7



Programmiersprachen als Träger von
Grundideen der Informatik

Jochen Ziegenbalg, Reutlingen

Programmiersprachen als Träger von Grundideen der Informatik¹

VON JOCHEN ZIEGENBALG

Mit 3 Abbildungen und 1 Tabelle

Programmiersprachen sind Werkzeuge zum Lösen von Problemen mit Hilfe von Computern. Sie sollten den Bedürfnissen des Benutzers möglichst gut angepaßt werden. Hierzu gehören insbesondere folgende Charakteristika:

- leichte Handhabbarkeit des verwendeten Sprachsystems (Interaktivität);
- Möglichkeiten zur Untergliederung und Strukturierung des Prozesses der Problemlösung (Strukturierbarkeit, Modularität);
- Möglichkeiten zur übersichtlichen Dokumentation des Programms (insbesondere auch zum Zweck der Kommunikation);
- Möglichkeiten, selbsterarbeitete Hilfsmittel (Modul) organisch in das Programmier-Grundsystem einzufügen (Erweiterbarkeit).

Im Hinblick auf diese Kriterien werden einige auf Mikro-Computern verfügbare Programmiersysteme (schwerpunktäßig: BASIC, PASCAL, LOGO, FORTH) diskutiert.

1 Vorbemerkungen

Die folgenden Überlegungen beziehen sich auf die Verwendung von (Mikro-)Computern im Unterricht oder in Lehr-/Lernsituationen. Auch wenn sich die professionelle Informatik mit Geräten ganz anderer Leistung beschäftigt, kommen beim Arbeiten mit Mikro-Computern bereits einige der fundamentalen Ideen und Methoden der Informatik zur Anwendung. (Ganz abgesehen davon, daß die Mikro-Computer von heute die Kapazität der Großrechner von gestern haben.)

Fast alle derzeit verfügbaren Mikro-Computer sind BASIC-Maschinen in dem Sinne, daß man sie nach dem Einschalten sofort in der Programmiersprache BASIC betreiben kann; sie haben, wie man oft hört, »BASIC im ROM«. Wenn der Benutzer dann nach einiger Zeit mit anderen Programmiersprachen arbeiten will, entdeckt er, daß er manche Sprache ganz anders handhaben muß als das altvertraute BASIC. Während er beim Arbeiten mit BASIC die Programmzeilen einfach der Reihe nach eingab und das teilweise bestehende Programm stets auflisten und auch ausführen konnte, muß er z. B. beim Arbeiten mit dem UCSD-PASCAL-System zunächst einen »Editor« aufrufen, mit dessen Hilfe er das Programm schreiben

kann. Bevor er das geschriebene Programm laufen lassen kann, muß er es »compilieren« lassen, was meist einige Rückkehrschleifen in den Editor zur Behebung von syntaktischen Fehlern zur Folge hat. Nachdem das Programm getestet ist, benötigt er noch die Hilfe des »Filers«, um es auf der Diskette abzuspeichern. Im Gegensatz zu dem Arbeiten in der BASIC-Umgebung wird sich der Benutzer bewußt, daß er ein hierarchisch gegliedertes System vor sich hat, dessen einzelne Komponenten ihm jeweils nur eine eng begrenzte Auswahl von Systemkommandos zugänglich machen.

Er wird beim Arbeiten mit dem neuen System im allgemeinen Vor- und Nachteile entdecken. Bei der Benutzung des UCSD-PASCAL-Systems macht der Benutzer zunächst die Erfahrung, daß es sehr viel komplizierter zu handhaben ist als das BASIC-System, mit dem er bisher gearbeitet hat. Er sieht aber sehr bald, daß ihm das komplexere System auch handfeste Vorteile bringt, wie z. B. einen ordentlichen Editor, mit dem er seine Programme jetzt wenigstens übersichtlich gestalten kann. Dafür lernt er gern die vielen neuen Editor-Kommandos. Auch der »Filer« erscheint ihm bald viel professioneller als der des alten BASIC-Systems. Nachdem er so die ersten Anfangsschwierigkeiten überwunden hat, erscheint ihm das neue System bald als sehr viel mächtiger als das alte. Im Laufe der Zeit stellen sich aber einige Enttäuschungen ein. So wird das Compilieren der Programme immer lästiger, besonders bei umfangreichen Programmen. Einen kleinen Schock bekommt der Benutzer, wenn er feststellt, daß die numerische Genauigkeit des neuen und vermeintlich in jeder Beziehung besseren Systems sehr viel schlechter ist als die im alten System. Zudem muß er von mancher liebgewonnenen Sitte bzw. Unsitten Abschied nehmen: er kann zur Fehlersuche nicht mehr einfach sein Programm stoppen, um den Zustand der Maschine anzusehen. Er findet es lästig, daß man den Umfang von »Arrays« schon vor der Ausführung des Programms durch eine Konstante festlegen muß, daß Arrays also nicht »dynamisch« sind. Auch hat er plötzlich keine richtigen Dateien mit wahlfreiem Zugriff mehr (random access files). Er kann den gesamten Dialog mit dem Rechner nicht mehr wie früher durch einen einzigen Befehl zum Zwecke der Dokumentation auf einen Drucker »umlenken« und vieles mehr.

Unser Benutzer hat sich das neue, mächtigere System gekauft, weil er sich bemüht, auf dem laufenden zu bleiben und aufmerksam die Diskussion um

¹ Vortrag auf der 74. Hauptversammlung des Deutschen Vereins zur Förderung des mathematischen und naturwissenschaftlichen Unterrichts 1983 in Tübingen.

Vorteile, Nachteile und Unzulänglichkeiten von Programmiersprachen verfolgt. Nun merkt er plötzlich, daß die Programmiersprache und ihre Syntax nur einen Aspekt des Systemverhaltens darstellen. Vieles, was ihm die Benutzung des Computers besonders angenehm gemacht hat, war von der Diskussion um die Programmiersprachen gar nicht erfaßt worden. Er hat gelernt, daß Programmiersysteme nicht nur nach dem Kriterium der Programmiersprachen-Syntax zu beurteilen sind. Der Computer stellt für ihn ein Werkzeug, eine Arbeitsumgebung dar, die ganzheitlich zu bewerten ist.

Die im folgenden zu diskutierenden (und einer subjektiven Bewertung entstammenden) Aspekte wie Modularität/Strukturiertheit, Interaktivität, Erweiterbarkeit und Effizienz sind also nicht nur als Charakteristika von Programmiersprachen, sondern als Eigenschaften der jeweiligen Sprachumgebungen zu verstehen. Als Sprachumgebung wird dabei die Einheit aus Programmiersprache und dem Betriebssystem, in das die Sprache eingebettet ist, bezeichnet.

2 Verwendungsmöglichkeiten für Computer im Unterricht

Manche Leute benutzen Computer im Unterricht, ohne sie jemals zu programmieren. Sie verwenden ausschließlich vorhandene Programme (Software), die sie meist mit dem Computer gekauft haben. Im Unterrichtsbereich gibt es eine Software-Kategorie, die schon mehrfach totgesagt wurde und mit jeder neuen Computergeneration wieder zu sprühen beginnt: den »programmierten Unterricht« in Form von »Drill-and-practise-Programmen«. Diese Verwendungsform wird gelegentlich auch als computerunterstützter Unterricht (CUU) bezeichnet.

Der Klarheit halber sei hier festgehalten, daß sich der vorliegende Artikel nicht auf diese Verwendungsform von Computern bezieht. Im folgenden steht die Nutzung des Computers als Werkzeug zum Problemlösen im Vordergrund. Bei dieser Verwendungsweise kommt der Erstellung von Programmen eine zentrale Rolle zu. A. ENGEL spricht bei dieser Verwendungsweise des Computers vom computerorientierten Unterricht.

3 Drei Charaktere beim Arbeiten mit Computern

Typ I: Kaum, daß ein Problem formuliert ist, zieht es ihn an den Computer, und er beginnt, das Programm zu schreiben. Er ist nicht geneigt, das Umfeld des Problems zu sondieren und etwa Fragen der folgenden Art zu stellen:

- In welche Teilprobleme zerfällt das gestellte Problem?

- Können schon früher erarbeitete Hilfsmittel zur Lösung des vorliegenden Problems verwendet werden?
- Welche Teilprobleme des vorliegenden Problems haben einen universellen Charakter und sollten deshalb zur Erleichterung der späteren Verwendbarkeit in einer hinreichend allgemeinen Form gelöst werden?

Erste Versionen seiner Programme sind meist relativ schnell fertig. Nach und nach werden den Anfangsversionen noch verschiedene Modifikationen aufgepropft. Die Programme werden immer weniger lesbar. Wenn ein Programm nach einem halben Jahr noch einmal in etwas modifizierter Form benötigt wird, ist es meist besser, das Programm neu zu schreiben, als sich durch die alten Programmisten zu wühlen. Dieser Typ könnte als der eines »fröhlichen Drauflos-Wurstlers« bezeichnet werden. (Das »fröhlich« bezieht sich allerdings meist nur auf die Anfangsphase der Programmierung; später verkehrt es sich häufig in sein Gegenteil.)

Typ II: Er hält nichts davon, zu schnell am Computer loszuwursteln, sondern er geht systematisch an das Problem heran. Er ist geneigt, das Problem erst vollständig zu analysieren und eine theoretische Lösung zu erarbeiten, bevor er sie auf den Computer überträgt. Seine Problemlösungen sind sauber beschrieben und gut dokumentiert. Er führt selbst erste »Papier-und-Bleistift-Tests« durch, noch bevor er einen Computer berührt hat. (Bei der Übertragung auf die Maschine gibt es dann wegen der kleineren und auch größeren Widerwärtigkeiten realer Computer häufig Probleme, doch durch die heißt er sich zäh hindurch.) Ein experimentelles Arbeiten ist ihm suspekt. Er hat eine klare Vorstellung davon, was richtig ist und was nicht. Zum Beispiel ist ein Vorgehen nach der Top-down-Methode in Problemlöseprozessen stets richtig. Dieser Typus könnte etwa als der des »Systematikers« oder auch »Theoretikers« bezeichnet werden.

Typ III: Auch für ihn ist die Vorgehensweise von Typ I unakzeptabel. Aber er hält es nicht so lange bei Papier und Bleistift aus wie Typ II. Überhaupt: warum soll man eigentlich noch Papier und Bleistift benutzen, wenn man einen guten Bildschirm-Editor zur Verfügung hat? Fast alles, was er früher skizzenhaft auf Konzeptpapier geschrieben hat, schreibt er heute gleich am Bildschirm, dessen hervorragende Redigier- und Korrekturmöglichkeiten er kräftig ausnutzt. Für ihn ist der Bildschirm das neue Konzeptpapier. Wenn er also Texte jeder Art am Bildschirm entwirft, warum dann eigentlich nicht auch Programme oder Programmteile? Das beim Programmierung mit Papier und Bleistift notwendige spätere Eintippen der Programme schenkt

er sich gern. Wie Typ II liebt er es, seine Programme gut zu untergliedern und zu strukturieren. Aber er weiß, daß der Teufel im Detail stecken kann, und testet Teillösungen gern unabhängig von ihrer Einbindung in eine Gesamtlösung aus. Die Lösung seines Gesamtproblems entsteht langsam als ein relativ geordnet gewachsenes Konglomerat von einzelnen Teillösungen (Modulen). Er experimentiert gern mit den einzelnen Modulen, schreibt gelegentlich verschiedene Varianten eines Moduls und beobachtet, wie sie sich sowohl einzeln als auch im Rahmen der Gesamtlösung verhalten. Dieser Typ könnte etwa als der des planend vorgehenden Handwerkers oder Ingenieurs bezeichnet werden.

Jeder der soeben holzschnittartig beschriebenen Typen wird durch eine bestimmte Programmierumgebung gefördert oder gar provoziert. Durch Typ I wird ein häufig in BASIC-Systemen anzutreffender Arbeitsstil beschrieben; Typ II ist eher für das Arbeiten in PASCAL-Systemen charakteristisch. Gibt es Programmierumgebungen, die den dritten Arbeitsstil unterstützen? Welche Eigenschaften müßten solche Programmierumgebungen haben?

4 Einige zentrale Eigenschaften von Programmierumgebungen

4.1 Modularität

Unter diesem Prinzip versteht man, grob gesprochen, die Anwendung des »Baukasten-Prinzips« in Problemlöseprozessen. Das heißt insbesondere:

- Zerlegung eines komplexen Problems in einfache, überschaubare Bestandteile (Modulen). Jeder Modul soll nach Möglichkeit eine gewisse eigenständige Bedeutung haben.
- Bearbeitung und Lösung der einzelnen Teilprobleme.
- Zusammensetzen der Teillösungen zu einer Gesamtlösung des Ausgangsproblems. Dabei ist das Augenmerk sowohl in der Entwurfsphase als auch in der Realisierungsphase besonders auf die Schnittstellen zwischen den einzelnen Modulen zu legen.

Modulares Vorgehen bedeutet zugleich stets einen gegliederten Entwurf von Problemlösungen. Dies ist sowohl im Sinne der Top-down-Methode als auch nach der Bottom-up-Methode möglich. Modulare Lösungen sind in natürlicher Weise strukturiert. Das wirkt sich auch positiv auf die Dokumentation der Programme aus. Modulare Programme sind in weit höherem Maße selbstdokumentierend als nichtmodulare Programme.

Notwendige Bestandteile der modularen Sprache sind die algorithmische Abstraktion und die Datenabstraktion. Die Programmiersprache, in der eine modu-

lare Lösung formuliert wird, muß also über Prozeduren bzw. allgemeine (»mehrzeilige«) Funktionen mit möglichst wenig eingeschränkten Parameterübergabe-mechanismen und die Möglichkeiten zur Datenstrukturierung, also zur Definition neuer Datentypen, verfügen. Insbesondere sollten Funktionen als Parameter übergeben werden können (wir kommen später bei der Diskussion von LOGO auf diesen Punkt zurück). Innerhalb von Prozeduren und Funktionen sollten lokale Variable definierbar sein, so daß es nicht zu Namenskonflikten mit etwaigen gleichnamigen Variablen in anderen Prozeduren kommt. Prozeduren und Funktionen sollten sich selbst aufrufen können; das heißt, die Sprache sollte über Rekursion verfügen.

Zum Zwecke der besseren Lesbarkeit und Dokumentation sollten beliebig lange, sprechende Namen für Prozeduren, Funktionen und Variable erlaubt sein.

Ein Haupthindernis für die Lesbarkeit, Dokumentation und Korrektheitsüberprüfung von Programmen ist die Möglichkeit, wild im Programm hin und her zu springen. Gut dokumentierbare Sprachen müssen deshalb über sinnvolle, beliebig schachtelbare Kontrollstrukturen zur Vermeidung von Sprüngen verfügen. Auch die Rekursion kann als Kontrollstruktur angesehen werden.

Die Struktur eines Programmes sollte auch rein optisch erkennbar sein. Modularisierte Programmiersprachen müssen deshalb auch über formatfreie Programmieditor zur besseren Darstellung der Schachtelungstiefe von Kontrollstrukturen verfügen.

4.2 Interaktivität

Die Frage der Interaktivität von Programmierumgebungen ist bei der Diskussion um die Struktur von Programmiersprachen stark vernachlässigt worden.

Eines der wesentlichen Zielleiteraktiver Systeme ist, dem Benutzer eine möglichst große »freie Beweglichkeit« im System zu gewähren. Hindernis für die freie Beweglichkeit ist das Vorhandensein vieler verschiedener System-Modi, die der Benutzer explizit aufrufen und verlassen muß und in denen ihm jeweils nur ein stark eingeschränktes Repertoire von Kommandos zur Verfügung steht. Besonders verwirrend für den Anfänger ist die Tatsache, daß die Wirkung eines Befehls vom Arbeitsmodus abhängen kann, in dem er sich gerade befindet. Die Befürworter der Interaktivität tragen daher häufig den Schlachtruf »don't mode me in« auf ihrem Panier.

Ein Charakteristikum interaktiver Systeme ist die Verschmelzung von Programmiersprache und Betriebssystem. So ist es bei BASIC-Betriebssystemen im allgemeinen unbedeutend, ob man die Kommandos RUN, SAVE oder DELETE zur Programmiersprache

oder zum Betriebssystem rechnet. Jeder dieser Befehle ist i. allg. von jeder Stelle des Systems, also auch von Programmen aus, aufrufbar.

Besonders wichtig für die Nutzung des Computers als Werkzeug zum Problemlösen ist der leichte Übergang zwischen den Phasen des Editierens und der Ausführung von Programmen. Dazu gehört insbesondere, daß die einzelnen Modulen unabhängig voneinander editiert und ausgeführt werden können. Für die Testphase sollten Testhilfen, wie z. B. die Einzelschritt-Protokollierung (Trace), zur Verfügung stehen.

Bei der Bewertung der Frage, ob Variable grundsätzlich zu deklarieren sein sollten oder nicht, scheiden sich häufig die Geister. Nicht nur von Anfängern wird die Notwendigkeit der Variablen Deklaration als lästig empfunden, besonders bei kleinen Programmen oder Modulen, wo der Deklarationsteil denselben Umfang einnehmen kann wie das eigentliche Programm. Andererseits soll nicht verschwiegen werden, daß der Zwang zur Deklaration von Variablen Leichtsinn- oder Schreibfehler vermeiden hilft. Kritisch wird es, wenn durch die Variablen Deklaration Zuweisungsmöglichkeiten über Gebühr beschnitten werden. Besonders bei PASCAL-Systemen wird die restriktive Handhabung der Datentypisierung vielfach als zu starr empfunden. Insgesamt gehören zum unkomplizierten Umgang mit Daten auch noch die dynamische Speicherverwaltung und die wohlwollende Interpretation von Datentypen durch das System. (Wenn einer Variablen z. B. die Zahl 3,14 zugeordnet wird, soll das System von selbst erkennen, daß es sich um eine Dezimalzahlvariable handelt.)

Weitere Zielsetzungen interaktiver Systeme sind die Dialogfreundlichkeit, leichte Erlernbarkeit der Grundfunktionen des Systems sowie eine allgemeine »Benutzerfreundlichkeit«. Dazu gehört z. B. auch, daß das System bei Bedienungsfehlern »freundlich« reagiert, also den Fehler deutlich beschreibt, Wege zur Behebung des Fehlers aufzeigt und das fehlerhafte Verhalten so wenig wie möglich bestraft.

Zu dieser allgemeinen Benutzerfreundlichkeit gehört weiterhin, daß sich der gesamte Bildschirm-Dialog mit dem Computer leicht parallel auf einen Drucker oder ein anderes Peripheriegerät lenken läßt.

Interaktive Systeme arbeiten (mit Ausnahme der »threaded languages« wie FORTH) meist interpretierend. Sie sind deswegen in der Regel ziemlich langsam. Dies scheint mir in einer Lernsituation aber nicht besonders gravierend zu sein. Außerdem werden zusätzlich zu den Programmierspracheninterpretoren verstärkt Compiler geliefert, mit denen Programme, die vollständig durchgetestet sind, in laufzeit-effiziente Versionen übersetzt werden können.

4.3 Erweiterbarkeit

Hierunter versteht man die Tatsache, daß die erarbeiteten Modulen nahtlos in die bestehende Programmierumgebung eingegliedert werden können. Zwischen den Grundbefehlen des Systems und den hinzugefügten Befehlen soll nach Möglichkeit kein »Klassenunterschied« bestehen; sie sollen für Drittbenuutzer nicht unterscheidbar sein. In Unterrichtssituationen wird es dem Lehrenden dadurch ermöglicht, dem Lernenden »Mikrowelten« zur Verfügung zu stellen, in denen er sich experimentell betätigen kann. So könnte der Lehrende dem Schüler z. B. eine Bruchrechenumgebung oder eine bestimmte Geometrienumgebung zur Verfügung stellen.

Noch anspruchsvoller ist die Zielsetzung, daß sich der Lernende die jeweilige Umgebung selbst konstruiert. In Abwandlung der über D. HILBERT berichteten Anekdote, wonach man ein (mathematisches) Problem erst dann richtig verstanden habe, wenn man auf die Straße gehen und es einem beliebigen Passanten erklären könne, läßt sich die Zielsetzung dieser Unterrichtsphilosophie so beschreiben, daß jemand ein Problem dann verstanden hat, wenn er einem Computer die Lösung des Problems beibringen kann. Ein Schüler hat also z. B. dann die Bruchrechnung verstanden, wenn er der Programmierumgebung des Computers ein »Bruchrechenpaket« eingliedern kann.

Erweiterbare Programmierumgebungen ermöglichen also die beständige, benutzerorientierte Eingliederung von neuen »Brocken« in das Gesamtsystem. Dieser Vorgehensweise sind keine prinzipiellen Schranken, sondern nur faktische Beschränkungen durch die zur Verfügung stehende Hardware (z. B. Speicherplatz) gesetzt. Da die Arbeitsweise in erweiterbaren Systemen in der Erstellung eines beständig wachsenden Netzes miteinander kommunizierender Prozeduren besteht, ist Modularität eine notwendige Voraussetzung für Erweiterbarkeit.

Um das Problemlösen in kleinen Schritten durch Versuch und Irrtum (»inkrementelles Problemlösen«) zu ermöglichen, müssen die Prozeduren in erweiterbaren Sprachen einzeln editierbar und testbar sein. Die hierfür notwendige leichte Beweglichkeit des Benutzers in der Programmierumgebung setzt also ein hohes Maß an Interaktivität voraus.

Modularität und Interaktivität sind also unverzichtbare Bestandteile von erweiterbaren Systemen. Auf Mikro-Computern standen als Programmierumgebungen in der Anfangsphase praktisch nur BASIC-Systeme zur Verfügung. Sie waren zwar einigermaßen interaktiv (nicht zuletzt deswegen fanden die Mikro-Computer ja eine derart weite Verbreitung), die Modularität der Sprache war jedoch absolut unbefriedigend. Danach war ein gewisser PASCAL-Boom zu

verzeichnen. PASCAL-Systeme weisen zwar eine erheblich verbesserte, wenn auch nicht voll zufriedenstellende Modularität auf; sehr bald wurde jedoch allgemein (besonders im Vergleich zu den vorigen BASIC-Systemen) ein Mangel an Interaktivität beklagt.

Wären diese Eigenschaften unvereinbar, so müßte sich jeder Benutzer entweder für Modularität oder für Interaktivität entscheiden. Für mich persönlich hätte dann die Modularität den höheren Stellenwert. Glücklicherweise sind diese Eigenschaften aber durchaus vereinbar. Versuche zur Entwicklung sowohl modularer als auch interaktiver Systeme gehen von verschiedenen Ausgangspunkten aus:

- Es wird versucht, BASIC durch ein Prozedurkonzept bzw. allgemeines Funktionskonzept und reichhaltigere Kontrollstrukturen zu erweitern. Der neue ANSI-Standard geht in diese Richtung. In Großbritannien ist man in diesem Punkte besonders weit vorangeschritten. Der BBC-Mikrocomputer enthält im ROM schon lange ein sehr vernünftiges BASIC mit Prozeduren, Funktionen, verbesserten Kontrollstrukturen und ziemlich freier Editierbarkeit. Die Firma U-Microcomputers in Warrington (Cheshire) hat für andere Mikro-Computer ein sehr komfortables strukturiertes BASIC entwickelt. In Dänemark hat B. CHRISTENSEN die Sprache COMAL (COMmon Algorithmic Language) entwickelt, die über ähnliche Eigenschaften wie die strukturierten BASICs verfügt. Im Hinblick auf die algorithmische Strukturierbarkeit von BASIC tut sich also einiges; die Datenstrukturierung wurde allerdings (noch?) nicht angepackt. Diese neueren Sprachversionen haben allerdings so wenig mit den alten BASIC-Versionen gemeinsam, daß man sich überlegen sollte, ob man ihnen nicht doch einen neuen Namen gibt. Der Name COMAL scheint mir sehr treffend.
- Komplementär zu der soeben geschilderten Entwicklung bei BASIC gibt es Versuche, interaktionsfreundlichere Versionen von PASCAL herauszubringen.
- Schließlich gibt es neuere Sprachentwicklungen, bei denen Modularität und Interaktivität von vornherein im Sprachdesign verankert sind und nicht erst nachträglich aufgepflanzt wurden. Hierzu rechne ich vor allem die LISP/LOGO-Sprachsysteme; im weiteren Sinne aber auch die Sprache FORTH.
- Über ELAN möchte ich an dieser Stelle nichts sagen, da mir keine einigermaßen vollständige Version für die gängigen Mikro-Computer bekannt ist.

Da LOGO- und FORTH-Systeme weit weniger verbreitet sind als BASIC- und PASCAL-Systeme, möchte ich im folgenden Abschnitt einige Aspekte der

Systemerweiterbarkeit am Beispiel dieser beiden Sprachen erläutern.

5 Aspekte der Spracherweiterung am Beispiel von LOGO

Die Entwicklung von LOGO begann etwa ab 1968 im Artificial Intelligence Laboratory des Massachusetts Institute of Technology (MIT) unter der Leitung von S. PAPERT. Weil (nicht obwohl!) LOGO als eine Programmiersprache für Kinder entworfen wurde, mußte sie nach dem Willen ihrer Entwickler eine äußerst mächtige Sprache sein; nicht etwa eine »Spielzeugsprache«, wie manch einer zu denken geneigt ist, wenn er »Programmiersprache für Kinder« hört. LOGO war deshalb über sehr lange Zeit nur auf sehr großen Rechnern verfügbar. Die Implementierung auf Mikrocomputer kam erst relativ spät, etwa in den Jahren 1981/82. Sie wurde unter der Leitung von H. ABELSON (MIT) durchgeführt.

Hauptcharakteristika von LOGO sind:

- Sehr große Modularität durch ein volles Prozedurkonzept. In LOGO gibt es keinen Unterschied zwischen Prozeduren und (mehrzeiligen) Funktionen. Sowohl als Eingabeparameter als auch als Funktionswerte können beliebige Datentypen auftreten.
- LOGO basiert sehr stark auf der Idee der Rekursion. Rekursive Prozeduren und Datentypen sind möglich. Rekursive Prozeduren sind jedoch meist nicht sehr laufzeit-effizient, und sie benötigen i. allg. viel Speicherplatz. Prozeduren mit der sogenannten »tail recursion« oder »last line recursion«, wo also der rekursive Aufruf nur einmal und zwar als letzter Befehl einer Prozedur vorkommt, werden intern allerdings in iterative Versionen überführt. Solche rekursiven Prozeduren können dann z. B. beliebig lange laufen (siehe z. B. die SOLANGE-Prozedur). Dies stellt eine erheblich bessere Implementierung rekursiver Prozeduren als in vielen anderen Programmiersprachen dar.
- Die Erweiterbarkeit der Sprache ist einer der Ecksteine ihres Designs. Benutzerdefinierte LOGO-Prozeduren sind exakt wie LOGO-Grundwörter verwendbar. Selbst neue Kontrollstrukturen können dem LOGO-System leicht hinzugefügt werden (siehe z. B. die SOLANGE-Prozedur).
- Daten sind in LOGO durch das beliebig hierarchisierbare Listen-Konzept strukturierbar. Die Allgemeinheit und Flexibilität dieser Methode wird klar, wenn man sich deutlich macht, daß Listen ebenso universell sind wie Bäume (im Sinne der Graphentheorie). Auch Programme werden als Listen gespeichert; sie sind mit den üblichen Listenverarbeitungsbefehlen beliebig manipulierbar.

- In LOGO wird konsequent zwischen dem Namen und dem Wert einer Variablen unterschieden. Diese bei den meisten Programmiersprachen fehlende Unterscheidung macht natürliche, elegante und effiziente Problemlösungen möglich.
- LOGO enthält ein Paket von geometrisch orientierten Befehlen, die sich wegen ihrer Körperzentriertheit besonders für einen informellen Einstieg in geometrische Fragen eignen. Diese Art von Geometrie, die sogenannte »turtle geometry«, hat weit über LOGO hinaus an Bedeutung gewonnen.

Nach allem, was bisher gesagt wurde, dürfte dem Kenner von LISP die große Ähnlichkeit zwischen LISP und LOGO auffallen. In der Tat ist LOGO ein Dialekt der LISP-Familie. Die Hauptunterschiede zwischen LISP und LOGO sind:

- Die LOGO-Grundwörter sind stärker an der Umgangssprache orientiert (z. B. FIRST statt CAR).
- Im Bereich der Arithmetik wird in LOGO die Infix-Notation verwendet; z. B.: $2 + 3$ an Stelle von $(+ 2 3)$.
- In LOGO benötigt man im allgemeinen weit weniger Klammern als in LISP.
- Es gibt eine Version von LOGO mit deutschen Grundwörtern, Fehlermeldungen und Kommentaren.

5.1 Das Variablenkonzept von LOGO

Mit "X wird in LOGO der Name und mit WERT "X der Wert der Variablen X bezeichnet. An Stelle von WERT "X kann man auch kurz :X schreiben.

Beispiele

(Mit → soll die »Antwort« des Computers gekennzeichnet werden.)

```
SETZE "A "B
DRUCKE WERT "A
→ B
SETZE "B "C
DRUCKE WERT WERT "A
→ C
SETZE WERT "B "D
DRUCKE WERT "C
→ D
SETZE WERT "C WERT "A
DRUCKE WERT "D
→ B
```

Anwendungsbeispiel: Wörterbuch

In den meisten Programmiersprachen wird man derartige Probleme mit einer zweidimensionalen Tabelle und den dazugehörigen Such- und Speicherprogrammen lösen. Auch in LOGO könnte man diese Me-

thode anwenden. Das Variablenkonzept von LOGO läßt aber auch die folgende natürliche, elegante und effiziente Lösung zu:

```
SETZE "APFEL "APPLE
SETZE "BAUM "TREE
SETZE "HAUS "HOUSE
SETZE "TISCH "TABLE
...
DRUCKE WERT "APFEL
→ APPLE
DRUCKE WERT "HAUS
→ HOUSE
...
SETZE "APPLE "APFEL
...
DRUCKE WERT "APPLE
→ APFEL
DRUCKE WERT WERT "APFEL
→ APFEL
```

Vom Programmierer zu schreibende Tabellen-Suchroutinen entfallen vollständig. Die Rückgabe eines gesuchten Wortes erfolgt blitzschnell. Dieses Beispiel zeigt, daß LOGO nicht grundsätzlich als langsame Sprache anzusehen ist. Durch Ausnutzung LOGO-spezifischer Eigenschaften lassen sich häufig sehr natürliche und sehr laufzeit-effiziente Lösungen finden.

5.2 Listen und LOGO

Ohne allzu detailliert in das Listenkonzept einzusteigen, soll hier nur kurz skizziert werden, daß Listen in (binäre) Bäume überführt werden können und wie man endliche Bäume in Listen überführen kann.

Listen werden syntaktisch durch eckige Klammern gekennzeichnet. Sie sind beliebig schachtelbar. Ein Beispiel:

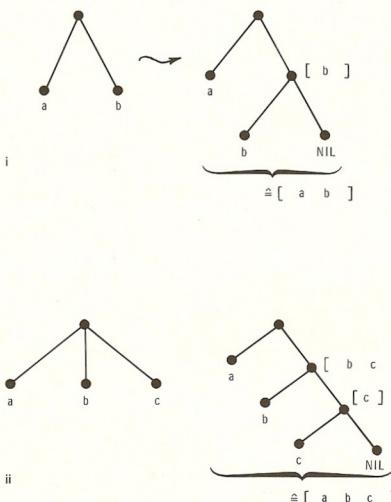
[Dies ist [eine Liste [aus]] [4 Elementen.]]

Der Baum in Abbildung 1 zeigt, wie die Listenverarbeitungsfunktionen ERSTES (kurz: ER) und OHNEERSTES (kurz: OE) arbeiten. Umgekehrt lassen sich auch Bäume in Listen verwandeln (Abb. 2).

5.3 Eine kleine Studie in Interaktivität und Modularität von LOGO

Problem: Wertetafeln von Funktionen

Es gibt eine Reihe von Problemen, bei denen es wünschenswert ist, daß man Funktionen als Parameter übergeben kann, z. B. Differentiation und Integration von Funktionen, Funktionsschaubilder oder auch nur Wertetafeln von Funktionen. Bei den meisten Programmiersprachen ist eine Übergabe von Funktionen

Abb. 1. \triangle Liste \rightarrow (binärer) Baum. NIL = []: leere ListeAbb. 2. ∇ (i) Binärer Baum \rightarrow Liste, (ii) Nichtbinärer Baum \rightarrow Liste

als Ein- oder Ausgabeparameter von Prozeduren nicht möglich. Man muß dann für jede zu behandelnde Funktion ein eigenes Programm schreiben. In BASIC muß man dazu zumindest die Zeile, in der die Funktion definiert ist, neu schreiben. In PASCAL kommt noch ein weiterer Compilierungslauf hinzu.

Am folgenden Beispiel soll verdeutlicht werden, daß man in LOGO beliebige Funktionen als Parameter übergeben kann. In derselben Weise kann man übrigens auch ganze Programme als Parameter übergeben und durch Prozeduren modifizieren lassen (siehe z. B. [6]).

Beim modularen Arbeiten tritt stets die Frage auf, was die »natürlichen« Parameter eines bestimmten Problems sind. Bei Problem »Wertetafel« ist dies ziemlich klar:

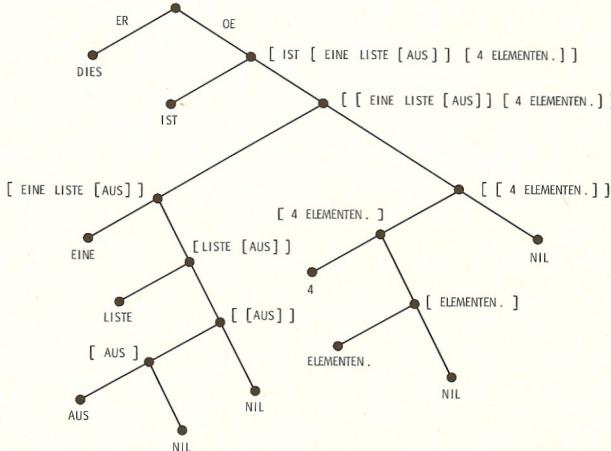
- die Funktion,
- die untere Schranke,
- die obere Schranke,
- die Schrittweite.

Diese Parameter finden sich in der Parameterliste der Prozedur WERTETAFEL wieder:

```
PR  WERTETAFEL :FKT :VON :BIS :SCHR
    SETZE "X :VON
    WENN :X > :BIS DANN RUECKKEHR
    DRUCKE :X
    DRUCKE TUE :FKT
    WERTETAFEL :FKT (:VON + :SCHR)
        ;BIS :SCHR
```

ENDE

[DIES IST [EINE LISTE [AUS]] [4 ELEMENTEN.]]



Die Prozedur ist weitgehend selbsterklärend. Dennoch ein paar Bemerkungen. "X" ist der Name des Funktionsarguments. RUECKKEHR (kurz: RK) bewirkt das Verlassen der Prozedur (zur aufrufenden Prozedur oder zum »top level« des Interpreters). Der Befehl TUE erzwingt die Auswertung einer eingegebenen Liste; im obigen Fall: die Auswertung der fraglichen Funktion an der Stelle :X. Der letzte Befehl in der WERTETAFEL-Prozedur ist der rekursive Aufruf der Prozedur selbst, wobei die untere Schranke allerdings um die Schrittweite erhöht ist. Dies ist ein Beispiel für »tail recursion«, das intern in eine Iteration übersetzt wird. Die Prozedur kann also im Prinzip beliebig lange laufen. In den meisten anderen Sprachen ist eine analoge Vorgehensweise nicht möglich, weil

- die Sprache nicht über Rekursion verfügt,
- Funktionen nicht als Parameter übergeben werden können,
- selbst beim Vorliegen von Rekursion die »tail recursion« nicht in eine Iteration übersetzt wird.

Beispiele für den Aufruf der obigen Prozedur (sei es direkt von der Tastatur aus, sei es von einem »Hauptprogramm«, sei es von einer beliebigen anderen Prozedur aus):

```
WERTETAFEL [ 3*:X + 10 ] Ø 20 1
oder
WERTETAFEL [ 2*SIN(:X*:X) + COS(:X)
  - :X + 24 ] (-1) 5 0.2
...
...
```

5.4 Neue Kontrollstrukturen in LOGO

LOGO hat nicht sehr viele Grundbefehle. Aufgrund seiner Erweiterbarkeit kann man aber diejenigen Befehle, die man gern benutzt, in maßgeschneiderter Form zum Grundsystem hinzufügen. Ich benutze z.B. gern die Kontrollstruktur SOLANGE (WHILE), die nicht im Grundrepertoire von LOGO enthalten ist. Die folgende Prozedur behebt diesen Mangel:

```
PR  SOLANGE :BED :HANDL
  PRUEFE TUE :BED
  WENNFALSCH DANN RUECKKEHR
  TUE :HANDL
  SOLANGE :BED :HANDL
ENDE
```

Beispiel für einen Aufruf:

```
SETZE "X Ø
SOLANGE [ :X < 100 ] [ DRUCKE :X*:X
  SETZE "X :X + 1 ]
```

BED ist der Name einer (als Liste zu spezifizierenden) Bedingung; HANDL der Name der jeweils aus-

zuführenden Handlung. Durch TUE :BED wird die Bedingungen ausgewertet. PRUEFE bereitet das entsprechende Ergebnis für die darauffolgende Abfrage WENNFALSCH (kurz WF) auf. Ist die Bedingung nicht erfüllt, so wird die Kontrolle an die aufrufende Prozedur übergeben, andernfalls bewirkt der Befehl TUE :HANDL, daß die spezifizierte Handlung ausgeführt wird. Der letzte Befehl ist wieder der tail-recursive Aufruf der Prozedur SOLANGE selbst; auch diese Prozedur kann also im Prinzip beliebig lange laufen.

LOGO enthält auch keine Zählschleifen mit spezifizierbarer Schrittweite; also das, was in BASIC meist als FOR ... NEXT Schleife implementiert ist. (Einfache Wiederholungsschleifen mit Schrittweite 1 sind in der Form der WIEDERHOLE-Anweisung vorhanden.)

Das nächste Beispiel einer »FUER«-Schleife dürfte nach den bisherigen Ausführungen selbsterklärend sein:

```
PR  FUER :LAUFVARIABLE :VON :BIS
  :SCHRITT :HANDL
  SETZE :LAUFVARIABLE :VON
    WENN WERT :LAUFVARIABLE > :BIS
    TUE :HANDL
    FUER :LAUFVARIABLE (:VON
      + :SCHRITT) :BIS :SCHRITT
      :HANDL
```

ENDE

Beispiel: FUER "K 1 26 2.5 [DRUCKE :K*:K]

LAUFVARIABLE ist der Name einer Variablen, deren Wert der Name einer weiteren Variablen (im obigen Fall K) ist.

Soviel zu LOGO. Um ein tieferes Verständnis für LOGO zu bekommen, muß man mit der Sprache experimentieren. Die LOGO-Sprachumgebung lädt dazu ein.

6 FORTH

Die Entwicklung von FORTH begann etwa ab 1970 durch CH. MOORE, der die folgende Zielsatzung beschrieb:

One principle that guided the evolution of FORTH . . . is bluntly: Keep it simple. A simple solution has elegance. . . . Simplicity provides confidence, reliability, compactness and speed.

L. BRODIE, der Autor des sehr populären Buches »Starting Forth«, formuliert:

What's the difference between FORTH and other high-level languages? To put it very briefly: it has to do with the compromise between man and computer. A language should be designed for the convenience of its human users, but at the

same time for compatibility with the operation of the computer. FORTH is unique among languages because its solution to this problem is unique.

Hauptcharakteristika von FORTH sind:

- Die Sprache ist sehr stark stack-orientiert. Es gibt einen Parameter-Stack für Daten und einen Return-Stack im wesentlichen (aber nicht ausschließlich) für Adressen. Die Stacks sind nur durch die Speichergröße beschränkt.
- Im Einklang mit dieser Stack-Orientiertheit ist die Syntax der Sprache fast durchweg auf der umgekehrten polnischen Notation (UPN) aufgebaut. Die Addition von zwei Zahlen geschieht z. B. folgendermaßen: 2 3 +
- Modulen werden in FORTH nicht als Prozeduren, sondern als Worte bezeichnet. Die Parameterübergabe wird mit Hilfe des Stacks durchgeführt. Dies schafft einerseits große Flexibilität (es kann praktisch alles an Parametern übergeben werden), aber für die Korrektheitsüberwachung der Parameterübergabe ist der Programmierer selbst verantwortlich. Rekursion ist (auf direktem Wege) nicht möglich; sie kann mit Hilfe des Return-Stacks aber relativ leicht simuliert werden.
- Das Sprachdesign ist auf Erweiterbarkeit hin angelegt. FORTH-Programmierer sehen ihre Tätigkeit nicht so sehr als ein Schreiben von Programmen, sondern eher als das beständige Erweitern eines vorliegenden Wörterbuches (dictionary) an.
- Die Sprache verfügt über eine vergleichsweise gute Interaktivität. Neudefinierte FORTH-Worte können entweder im Zeileneditor geschrieben und sofort lokal compiliert oder in einem vollwertigen Bildschirmeditor geschrieben und danach compiliert werden. Nur im letzteren Fall ist ein FORTH-Wort editierbar. Bei der Compilierung wird jedes Wort zugleich das aktive »dictionary« eingefügt, das auch als Stack organisiert ist. Aufgrund dieser Prozedur bei der Generierung neuer Wörter zählt man FORTH zu den sogenannten »threaded languages«.
- Da FORTH-Wörter in compilierter Form ausgeführt werden und nicht interpretiert zu werden brauchen, sind FORTH-Programme außerordentlich laufzeit-effizient. Bei vielen Problemen ist der Zeitbedarf von FORTH-Programmen eher mit dem von Assembler-Programmen als mit dem anderer höherer Programmiersprachen vergleichbar (siehe Tabelle mit einigen Laufzeitvergleichen).
- Der Code von FORTH-Programmen ist sehr kompakt; man findet FORTH relativ häufig auf kleinen Rechnern implementiert. FORTH-Programme können aber sehr leicht unlesbar werden. (Gelegentlich wird FORTH als eine »Write-only-Sprache« bezeichnet.)

FORTH kommt in bezug auf Modularität und Interaktivität meines Erachtens nicht ganz an LOGO heran; sein großes Plus liegt aber in seiner Schnelligkeit und Kompaktheit, die es für viele echten Anwendungen sehr interessant machen. Auch wenn in Zukunft die Rechner noch erheblich schneller werden sollten, wird die Relation zwischen FORTH und den anderen Sprachen voraussichtlich bestehen bleiben.

FORTH wurde im Gegensatz zu LOGO ausschließlich im Hinblick auf Kompatibilität und Laufzeit-effizienz, nicht aber für pädagogische Zwecke entwickelt. Ein direkter Vergleich der beiden Sprachen ist deshalb nicht angemessen. Selbst wenn FORTH nicht die erste Sprache im Erziehungsbereich sein sollte, weist sie doch Eigenschaften auf, die auch für diesen Bereich nicht uninteressant sind. Auch für jemanden, der im Bildungsbereich tätig ist, erscheint es also nicht unangemessen, die Sprache im Auge zu behalten.

Da ich hier nicht zu tief in die Sprache einsteigen kann, will ich es zum Zwecke der Demonstration bei einer Version des Euklidischen Algorithmus bewenden lassen. Diese Version arbeitet nur mit dem Stack. Man kann in FORTH auch mit Variablen arbeiten. Dies führt zwar meist zu einer besseren Lesbarkeit der Programme, durch das häufigere Hin-und-her-Transportieren der Daten sind solche Programme aber langsamer.

```
: GGT BEGIN OVER OVER
    > IF SWAP THEN
        OVER —
        DUP
    0= UNTIL
    DROP ;
```

Aufruf: a b GGT
also z.B.: 18 24 GGT

Anstelle eines verbalen Kommentars begnügen ich mich mit dem »Film« des Stack (Abb. 3).

7 Ein Laufzeit-Vergleich

Der Vergleich bezieht sich auf drei Test-Programme:

- reine Zählschleife (von 1 bis 30 000) ohne bzw. mit Ausdruck der Laufvariablen,
- Zufallsgraphik: d. h. vom gegenwärtigen Punkt aus wird eine Verbindungsstrecke zu einem zufällig ausgewählten Bildschirm-Punkt gezogen (10 000 Wiederholungen),
- Euklidischer Algorithmus (GGT (10 000,1)), absichtlich in der langsamsten Form der Wechselwegnahme.

Die LOGO-Prozeduren sind jeweils einmal in iterativer und einmal in tail-rekursiver Form geschrieben. Bei PASCAL war Rekursion nicht möglich, da der

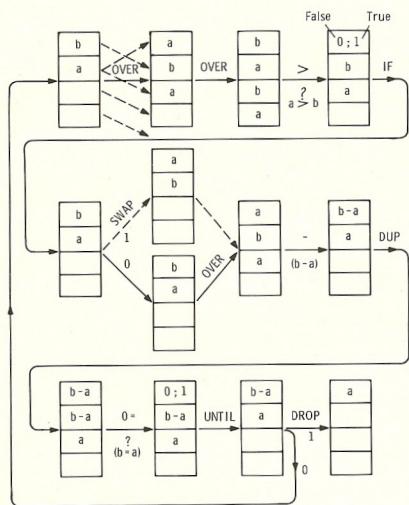


Abb. 3. »Film« des Stack

Aufbau des Rekursionsstacks den Speicher nach Größebenordnungsmäßig etwa 100 Aufrufen sprengt.

Die Laufzeitunterschiede sind enorm (Tab. 1). Es fällt auf, daß FORTH durch den Ausdruck außerordentlich stark »gebremst« wird.

Testprogramme

(A) Zählschleife

Die Programme stellen in der verwendeten Programmiersprache jeweils die minimale Version dar.

```
BASIC: 10 INPUT N
        20 FOR I=1 TO N : NEXT I
        30 END
```

```
PASCAL: PROGRAM COUNT;
        VAR I ,N: INTEGER;
        BEGIN
          READLN(N);
          FOR I:=1 TO N DO;
        END.
```

```
LOGO iterativ:
PR ZAEHLEN :N
  WIEDERHOLE :N []
ENDE
Aufruf: ZAEHLEN 30 000
```

	Zählschleife (1, ..., 30 000)	Zufalls- graphik (1, ..., 10 000)	GGT (10 000, 1) (Subtraktionsform)
	ohne Ausdruck	mit Ausdruck	
BASIC	41 s	14 min 9 s	5 min 32 s
FORTH	4 s	23 min 24 s	2 min 51 s
LOGO (iterativ)	4 min 24 s	26 min 54 s	15 min 28 s
LOGO (tail-rek.)	8 min 35 s	31 min 15 s	18 min 1 s
PASCAL	19 s	36 min 33 s	3 min 24 s

Tab. 1. Testergebnisse: Laufzeiten

LOGO tail-rekursiv:

```
PR ZAEHLEN :N
  WENN :N = Ø DANN
    RUECKKEHR
    ZAEHLEN (:N - 1)
  ENDE
```

Aufruf: ZAEHLEN 30 000

FORTH: : CNT DO UNTIL ;

Aufruf: 30 000 1 CNT

(B) Zufallsgraphik

```
BASIC: 10 INPUT N
        20 HGR2
        30 HPLOT 0,0
        40 FOR I = 1 TO N
        50 HPLOT TO RND(1)*279,
          RND(1)*159
        60 NEXT I
        70 END
```

```
PASCAL: PROGRAM ZUFALLSGRAPHIK;
        USES TURTLEGRAPHICS;
        VAR I ,N: INTEGER;
        BEGIN
          READLN(N);
          RANDOMIZE;
          INITTURTLE;
          PENCOLOR(WHITE);
          FOR I := 1 TO N DO
            MOVETO(RANDOM MOD 279,
              RANDOM MOD 159)
          END.
```

```
LOGO iterativ:
PR ZUFALLSGRAPHIK :N
  BILD
  VERSTECKIGEL
  WIEDERHOLE :N [ AUFXY
```

```

ZUFALLSZAHL 279
ZUFALLSZAHL 159 ]
ENDE
Aufruf: ZUFALLSGRAPHIK 10 000

LOGO tail-rekursiv:
PR ZUFALLSGRAPHIK :N
  WENN :N=0 DANN RUECKKEHR
  AUFXY ZUFALLSZAHL 279
  ZUFALLSZAHL 159
  ZUFALLSGRAPHIK (:N - 1)
ENDE
Aufruf: BILD VERSTECKIGEL
        ZUFALLSGRAPHIK 10 000

FORTH: : ZUFALLSGRAPHIK
HGR
FULL
DO 279 RANDOM 159 RANDOM
    LINETO LOOP ;
Aufruf: 10 000 1 ZUFALLSGRAPHIK

(C) Euklidischer Algorithmus

BASIC: 10 INPUT A
20 INPUT B
30 IF A = B THEN PRINT A : END
40 IF A > B THEN LET A = A - B
50 IF B > A THEN LET B = B - A
60 GOTO 30

PASCAL: PROGRAM GGT;
VAR A,B,R: INTEGER;
BEGIN
  READLN(A);
  READLN(B);
  REPEAT
    R := A;
    WHILE B <= R DO R := R - B;
    A := B;
    B := R;
  UNTIL R = 0;
  WRITELN(A);
END.

LOGO: PR GGT :A :B
      WENN :A < :B RUECKGABE
          GGT :A (:B - :A)
      WENN :B < :A RUECKGABE
          GGT (:A - :B) :B
      WENN :A = :B RUECKGABE :A
ENDE
Aufruf: DRUCKE GGT 10 000 1

FORTH: siehe oben.

```

8 Schlußbemerkungen

Bei der Diskussion von Programmiersprachen hört man häufig das Argument: »Aber das kann ich doch in der Sprache X auch machen!«. Dieser Einwand geht am Kern der Sache vorbei, denn man kann mit fast jeder Sprache praktisch alles machen. Im pädagogischen Kontext stehen dagegen die Fragen der Angemessenheit (insbesondere Natürlichkeit und Klarheit) der Formulierung und des Stils im Vordergrund.

Vom Standpunkt der Didaktik der Mathematik aus gesehen, sind Computer für mich vorrangig Werkzeuge im Prozeß des Problemlösens; und zwar sehr universelle Werkzeuge. Einen wesentlichen methodologischen Beitrag des Arbeitens mit Computern sehe ich in der Förderung eines experimentierfreudigen, entdeckenden (explorativen), den Bestand des Wissens und der Fähigkeiten in kleinen Schritten ständig erweiternden (inkrementellen) Arbeitsstils.

Ein solcher Arbeitsstil hat für mich viel mit einem handwerklichen Arbeitsstil gemeinsam. Man spricht ja nicht von ungefähr vom »Schnitzen« an Programmen. (Aufgrund dieser handwerklichen Charakteristik halte ich es übrigens für verfehlt, die Verwendung von Computern auf den gymnasialen Unterricht beschränken zu wollen. Gerade der intellektuellen Disposition von Haupt- und Realschülern kommt dieser experimentelle, handwerkliche Arbeitsstil durchaus zugestatten.)

Experimentierfreudigkeit und exploratives Arbeiten werden stark durch interaktive Sprachumgebungen gefördert. Interaktivität scheint mir also aus pädagogisch/didaktischer Sicht eine erstrebenswerte Eigenschaft von Sprachumgebungen zu sein. Damit das experimentelle Arbeiten aber nicht im Chaos versandet, muß die Programmiersprache über geeignete Ordnungs- und Gliederungsmittel verfügen. Dies wird im wesentlichen durch die Modularität der Sprache erreicht. Schließlich setzt beständiges Anwachsen der erarbeiteten Fähigkeiten die Erweiterbarkeit der Sprachumgebung voraus. In Werbeanzeigen liest man oft Anzeigen der Art: »Jetzt neue Superversior der Sprache X verfügbar: 20 neue Befehle!«. Nicht die zwanzig neuen Befehle sind wichtig, denn irgendwann benötigt man doch den 21. Befehl, der doch noch fehlt, sondern Erweiterbarkeit der Sprache ist gefragt, die es möglich macht, daß man die fehlenden Modulen den eigenen Bedürfnissen entsprechend in die Programmierumgebung einfügt.

Literatur

- [1] H. ABELSON: Logo for the Apple II. – Petersborough, New Hampshire: Byte/McGraw-Hill 1982.
Übersetzt und bearbeitet von H. LÖTHE unter dem

- Titel: Einführung in LOGO. – Vaterstetten: IWT Verlag 1983.
- [2] L. BRODIE: Starting Forth. – Englewood Cliffs, New Jersey: Prentice-Hall 1981.
- [3] Byte-Heft 8, 1980: FORTH. – Petersborough NH (USA): McGraw-Hill.
- [4] Byte-Heft 8, 1981: LOGO. – Petersborough NH (USA): McGraw-Hill.
- [5] S. PAPERT: Mindstorms. Kinder, Computer und neues Lernen. – Basel: Birkhäuser 1982.
- [6] J. ZIEGENBALG: Programm-Übersetzung mit LOGO. – LOG IN 4 (1984) 51.
-
- Anschrift des Verfassers: Prof. Dr. J. Ziegenbalg, Stelläckerstraße 17, 7410 Reutlingen

Aus der Schulpraxis · Für die Schulpraxis

Graphentheorie und Soziologie im Mathematikunterricht

Von GÜNTHER KARIGL

Mit 9 Abbildungen

Der vorliegende Beitrag, in dem einige einfache graphentheoretische Modelle aus der Soziologie behandelt werden, verfolgt eine zweifache Zielsetzung: Zum einen sollen Querverbindungen von der Mathematik zu anderen Wissenschaften (in diesem Fall zu den Sozialwissenschaften) aufgezeigt werden, zum anderen bietet sich die Möglichkeit, dem Schüler – ohne ihn zu überfordern – einen Einblick in neue Teilgebiete und Anwendungen der Mathematik zu gewähren. Im einzelnen erörtert werden die Repräsentation von sozialen Strukturen durch Soziogramme und gerichtete Graphen sowie eine Gleichgewichtstheorie in Graphen mit Vorzeichen. Ergänzend dazu sind im letzten Abschnitt einige Übungsaufgaben (einschließlich Lösungen) zusammengestellt, welche die konkrete Umsetzung dieses Themas im Unterricht erleichtern sollen.

1 Einleitung

Im Mittelpunkt des folgenden Beitrages, der als Einladung und Anregung zum anwendungsorientierten Unterricht anzusehen ist, steht ein Teilgebiet der Mathematik, nämlich die Graphentheorie, die sich in zahlreichen Problemstellungen als äußerst nützlich erwiesen hat und zudem gut geeignet ist, dem Schüler neue Aspekte der Mathematik näherzubringen.

Graphentheoretische Modelle sollen uns im folgenden – ähnlich wie in der Mengenlehre – nicht zu einer eigenen Theorie führen, sondern als mathematische Sprechweise zur Beschreibung soziologischer Phänomene dienen. Eine leicht lesbare Einführung in die Graphentheorie für den Lehrer stellt das Buch von

WILSON [6] dar. Der Schüler sollte diese Theorie jedoch nicht um ihrer selbst willen, sondern sogleich anhand von Beispielen kennenlernen. Dabei bietet sich eine Palette von Möglichkeiten aus den verschiedensten Anwendungsgebieten an. Dieser Artikel, der einige Beispiele und einfache Modelle aus der Soziologie behandelt, stellt dabei nur eine Alternative dar. Eine andere graphentheoretische Anwendung, die beispielsweise in die Schulphysik führt, wurde von MALLE [5] erörtert.

Ein Exkurs in die Graphentheorie könnte z. B. am Beginn der Sekundarstufe II unternommen werden, nämlich dann, wenn die verwandten Kapitel über Funktionen (und eventuell Relationen) im Unterricht behandelt werden. Im Verlauf der nächsten Abschnitte werden sämtliche Begriffe anhand von Musterbeispielen eingeführt oder zumindest erklärt. Weitere Aufgaben zum Üben und Vertiefen des Gelernten – z. B. in Form von Hausarbeiten – sind am Ende dieses Beitrages zusammengestellt und auch mit Lösungen versehen.

Die folgenden Abschnitte sollen somit als Vorschlag zur Auflockerung und Ergänzung des Mathematikunterrichts angesehen werden, sie dienen der Motivation des Schülers, vergrößern sein Interesse und seine Freude an der Mathematik und schaffen Querverbindungen zu anderen Wissenschaften. In diesem Zusammenhang sei auf den Artikel von KRÄNZER [3] hingewiesen, der jedem Lehrer der Mathematik empfohlen werden kann.