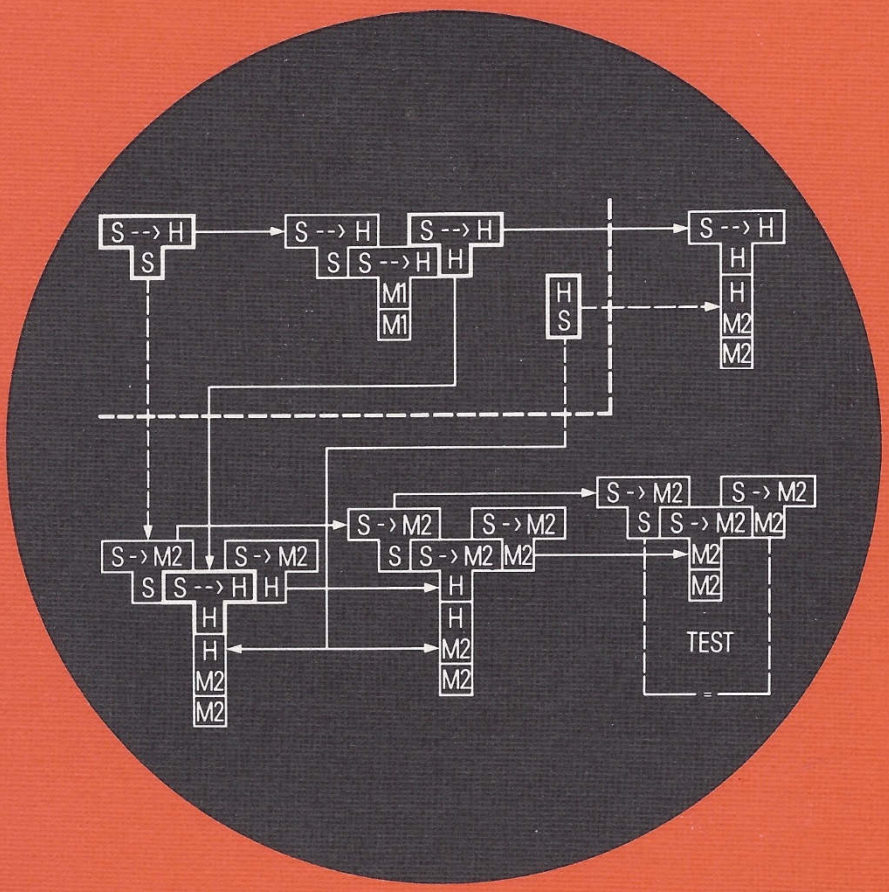
N. Wirth

Compilerbau



Teubner Studienbücher
Informatik

Leitfäden der angewandten Mathematik und Mechanik LAMM

Eine Einführung in die Grundlagen der formalen Sprachen und die Technik des Compilerbaus für Informatiker, Mathematiker und Ingenieure

Syntax und Satzzerlegung

Tabellengesteuerte Syntaxanalyse

Syntaxgraphen

Die Programmiersprache PL/O

Bau eines PL/O Compilers

Systematische Behandlung von syntaktischen Fehlern

Methodik der Erweiterung von Sprachen und Compilern

Technik der Compilerentwicklung und -übertragung

ISBN 3-519-32338-9

B.G.Teubner Stuttgart

Compilerbau

Eine Einführung

Von Dr. Dr. h. c. Niklaus Wirth
Professor an der Eidg. Technischen
Hochschule Zürich

4., durchgesehene Auflage
Mit 22 Figuren und 20 Übungen

B. G. Teubner Stuttgart 1986

Prof. Dr. Niklaus Wirth

Geboren 1934 in Winterthur, Schweiz. Von 1954 bis 1958 Studium an der Eidg. Technischen Hochschule Zürich mit Abschluß als Dipl. El.-Ing. Von 1959 bis 1960 Studium an der Université Laval, Quebec, Canada, und Erlangung des Grades M. Sc. Von 1960 bis 1963 Studium und anschließend Promotion an der University of California, Berkeley. Von 1963 bis 1967 Assistant Professor of Computer Science an der Stanford University. Von 1967 bis 1968 Assistenzprofessor an der Universität Zürich. Seit 1968 Professor für Informatik an der Eidg. Technischen Hochschule Zürich.

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Wirth, Niklaus:
Compilerbau : e. Einf. / von Niklaus Wirth. – 4.,
durchges. Aufl. – Stuttgart : Teubner, 1986.
(Leitfäden der angewandten Mathematik und Mechanik ;
Bd. 36) (Teubner-Studienbücher : Informatik)
ISBN 3-519-32338-9

NE: 1. GT

Printed in Germany
Gesamtherstellung: Beltz Offsetdruck, Hemsbach/Bergstraße
Umschlaggestaltung: W. Koch, Sindelfingen

Vorwort

Dieses Buch ist eine Einführung in die Theorie der formalen Sprachen und ihrer Satzzerlegung, sowie in die Technik der Compiler-Konstruktion. Diese Themen sind aus zweierlei Gründen relevant. Erstens erleichtert und vertieft die Kenntnis der Grundprinzipien eines Compilers das Verständnis für korrekte Verwendung von Programmiersprachen ganz allgemein. Zweitens ist die Beherrschung dieser Themen Voraussetzung für die fachgerechte Erstellung vieler Computer-Systeme, denen eine einfache Befehlssprache zugrunde liegt. Die Anzahl derartiger Anwendungen nimmt rasch zu, sei es im Gebiet der Prozesssteuerung, der Datenverarbeitung, oder der Betriebssysteme.

Bezüglich der Theorie der formalen Sprachen und der Methodik der syntaktischen Analyse beschränken wir uns auf das für den Bau eines einfachen Compilers minimal notwendige. Hingegen wird die systematische Entwicklung eines vollständigen Compilers für eine einfache Programmiersprache in ihren Einzelheiten dargelegt. Die Wahl der Quellsprache PL/0 ist ein Kompromiss zwischen einer Sprache, die allzu trivial wäre, um als lehrreiches Beispiel zu gelten, und einer Sprache, deren Mächtigkeit und Komplexität den wirklichen Kern und seine Hauptprinzipien verschleiern würde.

Für alle Programme wird die Sprache Pascal verwendet. Sie erlaubt eine transparente Darstellung komplexer Programme und Datenstrukturen, und sie eignet sich daher für diesen Themenkreis besonders gut.

Das Buch entstand aus einer Vorlesung an der ETH Zürich. Es ist eine stark überarbeitete Uebersetzung aus dem Englischen. Die Kapitel 12 und 13 wurden dabei neu hinzugefügt. Die Produktion des photo-reproduzierbaren Textes wurde vom Autor selbst vorgenommen. Dies war nur möglich unter Verwendung modernster Computersysteme für die Herstellung und Verarbeitung von Text und Illustrationen. Mein besonderer Dank gilt der Firma Xerox, die mir diese Werkzeuge in zuvorkommender Weise zur Verfügung gestellt hat.

Palo Alto, Dezember 1976

N. Wirth

Vorwort zur 3. Auflage

Das wesentliche Merkmal der vorliegenden, revidierten Fassung ist die Verwendung der Programmiersprache Modula-2 anstelle von Pascal zur Formulierung der dargelegten Algorithmen. Die damit zur Verfügung stehende Struktur des Moduls erweist sich als geradezu ideales Mittel, um die verschiedenen Teile eines Compiler-Systems auch thematisch zu trennen.

Der photo-kopierbare Text wurde am Arbeitsplatzrechner Lilith mit dem Dokumenten-Verarbeitungsprogramm Lara hergestellt. Dieses Paar hat mir die Arbeit ganz wesentlich erleichtert und zur Freude gemacht. Mein besonderer Dank gilt daher dessen Autoren J. Gutknecht und H. Schär.

Zürich, März 1984

N.W.

Vorwort zur 4. Auflage

In dieser vierten Auflage wurden lediglich einige Schreib- und Programmfehler ausgemerzt. Mein Dank geht an alle, die zu ihrer Auffindung beigetragen haben.

Zürich, Januar 1986 N.W.

Inhalt

0. Einleitung

Ziel dieses Buches ist die Entwicklung einer einfachen, rudimentären Programmiersprache und eines zugehörigen Compilers. Dieses Compiler-Programm ist ein Muster-Beispiel für eine systematische Entwicklung eines gut strukturierten Programmes nicht-trivialer Komplexität. Hauptsächlich aber soll dieses Kapitel als Einführung in die Technik des Compilerbaus dienen. Kenntnisse und Einblick in dieses Thema fördern ganz allgemein die Fähigkeiten des Programmierens in höheren Programmiersprachen. Ferner sind sie Grundlage und Vorbedingung für die Möglichkeit, eigene Eingabesprachen und Systeme für spezifische Zwecke zu konstruieren. Da der Compilerbau ein komplexes Thema mit vielen Aspekten ist, beschränken wir uns in dieser Hinsicht auf eine Einführung. Vielleicht die wichtigste Grundidee ist dabei die Erkenntnis, dass die grammatikalische Struktur einer Sprache sich in der Struktur des Compilers widerspiegeln muss. Daraus folgt unmittelbar, dass die Komplexität - oder besser die Einfachheit - einer Sprache das Mass für die Komplexität ihres Compilers darstellt. Wir beginnen daher mit einer Behandlung von Sprachstrukturen und deren formeller Beschreibung im allgemeinen. Danach konzentrieren wir uns ausschliesslich auf einfache Strukturelemente, die sich durch einfache, effiziente und übersichtliche Compiler behandeln lassen. Die Erfahrung hat gezeigt, dass Elemente von struktureller Einfachheit durchaus genügen, um praktisch alle echten Bedürfnisse zu erfüllen, die an Programmiersprachen gestellt werden.

Die zwei letzten Kapitel sind praktischen Problemen des Compiler-Baus gewidmet. Anhand einer Fall-Studie wird gezeigt, wie eine bestehende Sprache und ihr Compiler erweitert werden. Dieser in der Praxis häufige Fall wird durch eine systematische Struktur des zu erweiternden Systems ganz wesentlich erleichtert. Letzlich werden verschiedene Methoden der Übertragung eines Compilers auf fremde Computer dargelegt. Sie bauen alle auf der Technik des *bootstrapping* auf und bedingen, dass der Compiler in seiner eigenen Sprache beschrieben wird.

1. Definition und Struktur formaler Sprachen

Jeder Sprache liegt ein *Vokabular* zugrunde. Normalerweise bezeichnet man seine Elemente als Worte. Bei formalen Sprachen hingegen ist es üblich, sie *Symbole* oder Grundsymbole zu nennen. In jeder Sprache gibt es Folgen von diesen Symbolen, die als korrekt oder wohlgeformt, andere die als falsch oder missgebildet gelten. In erster Linie ist es die Grammatik oder *Syntax*, die bestimmt, zu welcher Kategorie eine Symbolfolge gehört. Wir gehen hier sogar soweit, dass wir die Menge von Symbolfolgen, die von der Syntax als wohlgeformt definiert sind, als die Sprache selbst bezeichnen. Missgebildete Folgen gehören überhaupt nicht zur Sprache, auch wenn sie ausschliesslich aus Symbolen des zugehörigen Vokabulars aufgebaut sind.

Die erste Funktion der Syntax ist also die Beschreibung der Menge der Symbolfolgen, genannt *Sätze*, die zur Sprache gehören. Die zweite, nicht minder wichtige Funktion ist die Definition einer Satzstruktur. Diese spielt eine eminente Rolle in der Erkennung der Bedeutung eines Satzes. Dies geht schon daraus hervor, dass Mehrdeutigkeiten meistens darauf beruhen, dass derselben Wortfolge mehrere korrekte Satzstrukturen zugeordnet werden können. Struktur (Syntax) und Bedeutung (Semanik) sind daher eng miteinander verbunden. Die Syntax dient letzlich immer einem höheren Ziel, nämlich einer Codierung einer bestimmten Bedeutung. Dies soll uns aber nicht daran hindern, zunächst ausschliesslich syntaktische Aspekte zu behandeln und die Probleme der Bedeutung und deren Interpretation zu ignorieren.

Zur Einführung wollen wir als Beispiel den Satz *Katzen schlafen* betrachten. Das Wort *Katzen* ist das Subjekt und *schlafen* das Praedikat des Satzes. Dieser Satz gehört zur Sprache, die zum Beispiel durch folgende Syntax definiert ist:

```
Satz     = Subjekt Praedikat.
Subjekt  = Katzen | Hunde.
Praedikat = essen | schlafen.
```

Die Bedeutung dieser drei Zeilen ist die folgende:

1. Ein Satz besteht aus einem Subjekt gefolgt von einem Praedikat.
2. Ein Subjekt besteht aus dem Wort Katzen oder dem Wort Hunde.

3. Ein Praedikat besteht aus dem Wort essen oder dem Wort schlafen.

Die Grundidee ist, dass jeder (wohlgeformte) Satz durch wiederholte Anwendung von Substitutionsregeln aus dem Startsymbol (*Satz*) hergeleitet werden kann. Als Notation verwenden wir im folgenden eine Variante der sogenannten Backus-Naur Form (BNF), die zuerst zur Definition der Programmiersprache Algol 60 verwendet wurde. Die Satzgefüge *Satz*, *Subjekt*, und *Praedikat* nennt man *nicht-terminale* Symbole, die Worte, die in den eigentlichen Sätzen vorkommen, heissen *terminale* Symbole, und die Substitutionsregeln heissen *Produktionen*. Die Zeichen =, | und der Punkt sind sogenannte *Metasymbole* der BNF Notation. Zur Benennung von nicht-terminalen Symbolen verwenden wir sinnvoll gewählte Worte, die auf die Bedeutung der entsprechenden Strukur hinweisen sollen. Wenn anstatt der deskriptiven Worte im obigen Beispiel lediglich einzelne Buchstaben eingesetzt werden, wie dies in mathematischen Formeln üblich ist, so erhalten wir folgende, formal beschriebene Syntax. Zur Unterscheidung verwenden wir Grossbuchstaben für nicht-terminale Symbole, Kleinbuchstaben für Terminalsymbole.

Beispiel 1

S = AB.
A = x | y.
B = z | w.

Die so definierte Sprache besteht aus den vier Sätzen xz, yz, xw, yw.

Die bisher informell eingeführten Konzepte sollen nun durch folgende Defintionen genau festgelegt werden. Wir bezeichnen einzelne Symbole durch lateinische, Symbolfolgen durch griechische Buchstaben.

1. Eine Sprache L wird charakterisiert durch ein Vier-Tupel L(T,N,P,S).

Das Vokabular T von Terminalsymbolen.
Die Menge N von nicht-terminalen Symbolen.
Die Menge P von Substitutionsregeln, genannt Produktionen.
Das Startsymbol S, ein Element von N.

2. Die Sprache L(T,N,P,S) ist die *Menge der Terminalsymbolfolgen*, die aus dem Startsymbol S nach den folgenden Regeln (siehe 3.) hergeleitet werden können.

$$L = \{\xi \mid S \overset{*}{\rightarrow} \xi \wedge \xi \in T^*\}$$

T^* bezeichnet die Menge der Symbolfolgen, die aus Symbolen des Vokabulars T aufgebaut werden können.

3. Eine Folge s_n kann aus einer Folge s_0 *hergeleitet* werden, wenn und nur wenn es Folgen $s_1, s_2, \ldots, s_{n-1}$ gibt, so dass jedes s_i aus dem vorangehenden s_{i-1} direkt hergeleitet werden kann (siehe 4.). Wir schreiben:

$$\sigma_0 \overset{*}{\rightarrow} \sigma_n \equiv \sigma_{i-1} \rightarrow \sigma_i \quad \text{für } i = 1 \ldots n$$

4. Eine Folge η kann aus einer Folge ξ *direkt hergeleitet* werden, wenn und nur wenn sich η und ξ als Folgen $\eta = \alpha\eta'\beta$ und $\xi = \alpha\xi'\beta$ darstellen lassen und $\eta' = \xi'$ eine Produktion in der Menge P ist. Man sagt, die Produktion finde im *Kontext* von α und β Anwendung.

Man beachte, dass die Notation $\alpha = \beta_1|\beta_2| \ldots |\beta_n$ lediglich eine Abkürzung für die Menge der Produktionen $\alpha = \beta_1, \alpha = \beta_2, \ldots, \alpha = \beta_n$ darstellt.

Nach diesen Ausführungen kann zum Beispiel der Satz xz aus Beispiel 1 durch folgende Reihe von Schritten hergeleitet werden:

$$S \rightarrow AB \rightarrow xB \rightarrow xz$$

Daraus folgt, dass $S \overset{*}{\rightarrow} xz$, und weil $xz \in T^*$, so ist xz ein Element der Sprache, d.h. $xz \in L$.

Man beachte, dass die nicht-terminalen Symbole A und B nur in Zwischenschritten vorkommen, und dass der letzte Schritt zu einer Folge führt, die nur terminale Symbole enthält. Die syntaktischen Regeln werden Produktionen genannt, weil sie bestimmen, wie ein Satz produziert werden kann.

Eine Sprache heisst (nach N. Chomsky) *kontext-frei,* wenn (und nur wenn) sie durch eine Menge von kontext-freien Produktionen definiert werden kann. Eine Produktion ist kontext-frei, wenn (und nur wenn) sie die Form

$$A = \sigma. \qquad (A \in N,\ \sigma \in (N \cup T)^*)$$

hat, das heisst, wenn ihre linke Seite aus einem einzigen Symbol besteht. Dies

bedeutet nichts weniger, als dass A durch σ ersetzt werden kann, ungeachtet des Kontextes, in dem A vorliegt. Falls eine Produktion die Form

$$\alpha A \beta = \alpha \sigma \beta.$$

hat, so heisst sie *kontext-abhängig*. Sie besagt nämlich, dass A nur durch σ ersetzt werden kann, sofern A in der Umgebung von α und β vorliegt. Wir werden uns im folgenden ausschliesslich mit kontext-freien Sprachen beschäftigen.

Das zweite Beispiel einer Syntax zeigt, wie durch die Verwendung von Rekursion eine unendliche Menge von Sätzen durch eine endliche Menge von Produktionen beschrieben werden kann.

Beispiel 2

S = xA.
A = z | yA.

Aus dem Startsymbol lassen sich die folgenden Sätze herleiten:

xz, xyz, xyyz, xyyyz, xyyyyz, xyyyyyz, ...

Das Beispiel zeigt also, wie durch Anwendung einer rekursiven Definition beliebige Repetitionen erzeugt werden können. Allerdings ist die Rekursivität nicht unmittelbar ersichtlich. Wir führen daher zur leichteren Sichtbarmachung einer Repetition eine zusätzliche Notation ein:

$$\{ \sigma \}$$

bedeute, dass die Folge σ beliebig viele Male (inklusive 0 mal) repetiert werden kann. Wenn die leere Folge mit e bezeichnet wird, so ist $\{\sigma\}$ gleichbedeutend mit ε | s | ss | sss | ... , und die zwei Produktionen aus Beispiel 2 lassen sich zusammenfassen in die eine Produktion

S = x{y}z.

wobei die Notwendigkeit für das Symbol A entfällt. Ebenfalls im Sinne einer Vereinfachung führen wir die Notation

$$[\sigma]$$

ein. $[\sigma]$ sei gleichbedeutend mit $\sigma \mid \varepsilon$. So besagt zum Beispiel die Produktion

S = x[y]z.

dass sich aus S die Folgen xyz und xz herleiten lassen. Es erscheint nun als angezeigt, auch gewöhnliche Klammerausdrücke zuzulassen. Anstelle von xyz | xwz, kann dann zum Beispiel x(y|w)z geschrieben werden. Es sei noch darauf hingewiesen, dass die Klammern { } [ ] ( ) Metasymbole wie die Zeichen = | und . sind. Sie gehören also zur erweiterten Metasprache, EBNF, und nicht zur Objektsprache, deren Syntax definiert wird.

2. Satzanalyse

Die Aufgabe eines Übersetzers oder Compilers ist normalerweise nicht die Erzeugung oder Herleitung, sondern die Erkennung eines Satzes und seiner Struktur. Dies bedeutet, dass die Schritte der Erzeugung beim Lesen eines Satzes als Schritte der Erkennung nachvollzogen werden müssen. Dies ist im allgemeinen eine komplizierte, und in vielen Fällen sogar eine unmögliche Aufgabe. Ihre Komplexität hängt unmittelbar von der Art der Produktionen ab, die die Syntax bestimmen. Zahlreiche Erkennungs-Algorithmen für verschiedene Klassen von Sprachen unterschiedlicher struktureller Komplexität sind bekannt. Ihre Effektivität hängt direkt von ihrer Mächtigkeit ab: je allgemeiner sie verwendbar sind, umso weniger Effizienz kann erwartet werden. Unser Ziel hier ist jedoch lediglich die Entwicklung einer Methodik, relativ einfach Sprachen zu erkennen, dies jedoch mit einer Effizienz, wie sie von der Praxis gefordert wird. Die Zeit, die gebraucht wird, um einen Satz zu erkennen, darf demnach höchstens proportional zu seiner Länge anwachsen. Anstatt einen möglichst allgemeinen Erkennungs-Algorithmus, genannt Parser, zu finden, gehen wir in umgekehrter Richtung vor: Wir postulieren einen einfachen Algorithmus und bestimmen danach die Klasse der Sprachen, die er verarbeiten kann.

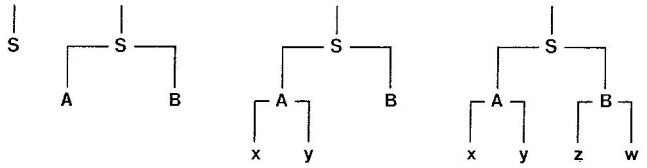
Eine erste Folge dieser grundlegenden Forderung ist, dass jeder Schritt in der Zerlegung eines Satzes lediglich durch den gegenwärtigen Zustand der Analyse und ein einziges, nächstes zu lesendes Symbol bestimmt werden darf. Ferner wird gefordert, dass kein Schritt später rückgängig gemacht werden darf (*one symbol lookahead, no backtracking*).

Die grundlegende Methodik, die nachfolgend beschrieben wird, beruht auf dem sogenannten *top-down* Prinzip. Es ist zweckmässig, die Struktur eines Satzes als Baum darzustellen. Zuoberst steht das Startsymbol, an seiner Basis der zu erkennende Satz. Zur Illustration möge folgendes Beispiel dienen:

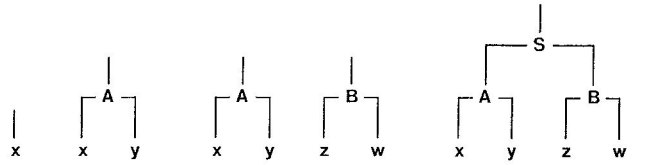
S = AB.
A = xy.
B = zw.

Beim Lesen (von links nach rechts) durchläuft der Parser die folgenden Schritte, wobei von S ausgehend eine Struktur aufgebaut wird, die an der Basis

dem zu lesenden Satz entspricht. Der Baum wächst also von oben nach unten; daher der Ausdruck top-down.



Umgekehrt wird beim sogenannten *bottom-up* Prizip vorgegangen. Anstatt "spekulativ" vom Startsymbol auszugehen und eine Folge zu produzieren, die dem eingelesenen Text entspricht, wird der eingelesene Text reduziert, bis zuletzt der ganze Text in das Startsymbol zurückgeführt ist. In der graphischen Darstellung wächst der Strukturbaum von unten nach oben; daher die Bezeichnung bottom-up.



Anhand des Beispiels 1 soll nun gezeigt werden, wie ein einfacher top-down Parser vorgeht. Wir nehmen an, dass er den Satz xw zu erkennen habe. xw gehört nur dann zur Sprache, wenn es aus dem Startsymbol S abgeleitet werden kann. Aus S ist jedoch nur die Folge AB direkt herleitbar. Wir ersetzen daher S auf jeden Fall durch AB. Nun muss untersucht werden, ob sich der Anfang des Satzes xw aus AB herleiten lässt. In der Tat bestätigt die Produktion A = x, dass dies möglich ist; x lässt sich also, zusammen mit A, als erledigt abstreichen. Es bleibt noch zu zeigen, dass w sich aus B ableiten lässt. Beim Durchsuchen der Syntax erkennen wir die Produktion B = z als nicht anwendbar; hingegen bringt uns die Produktion B = w ans Ziel. Wir können uns die einzelnen Schritte wie folgt aufzeichnen, wobei wir links die Symbolfolge auftragen, aus welcher der noch verbleibende Rest des zu erkennenden Satzes herleitbar sein muss. Rechts steht der Rest des Satzes selbst.

S xw

AB	xw
xB	xw
B	w
w	w

Man beachte, dass in diesem Beispiel jeder Schritt sich eindeutig aus dem zunächst zu verfolgenden Ziel (z.B. A) und dem nächsten vorliegenden Symbol (z.B. x) bestimmen lässt. Dies ist leider nicht in allen Fällen zutreffend, wie das folgende Beispiel einer Syntax zeigt.

Beispiel 3

S = A | B.
A = xA | y.
B = xB | z.

Wir versuchen, den Satz xxz (der tatsächlich zur definierten Sprache gehört) zu erkennen.

S	xxz
A	xxz
x	xxz
A	xz
xA	xz
A	z

An dieser Stelle zeigt es sich, dass es unmöglich ist, z aus A abzuleiten. Hätten wir im ersten Schritt S durch B anstatt durch A ersetzt, so wäre das Ziel erreichbar gewesen. Leider aber lässt sich in diesem Fall der Entscheid für A oder B nicht aus dem einen vorliegenden Eingabesymbol x ableiten. Nur ein Vorausblicken und Erkennen des Symbols z hätte den richtigen Entscheid ermöglicht. Es ist aber leicht ersichtlich, dass in diesem Beispiel einer Syntax keine feste Grenze für die Anzahl der zu betrachtenden Symbole gegeben werden kann. Denn jeder Algorithmus, der fähig wäre, über n Symbole vorauszublicken, könnte mit einer Folge von n Symbolen x gefolgt von einem z (oder y) zu Fall gebracht werden. Anstatt nun aber einen komplizierteren Erkennungs-Algorithmus vorzuschlagen, wollen wir eine Regel postulieren, die die Syntax einschränkt, indem sie den obigen Fall ausschliesst. Wir nehmen an, dass stets alle Produktionen mit demselben Linkssymbol in eine einzige Regel zusammengefasst vorliegen.

Definition 1

Wir bezeichnen die Menge aller Symbole, die am Anfang einer Folge stehen können, die aus einer Folge s herleitbar ist, mit first(s).

$$\text{first}(\sigma) = \{s \mid \sigma \dot{\rightarrow} s\sigma'\}$$

Regel 1

Für jede Produktion

$$A = \sigma_1 \mid \sigma_2 \mid \ldots \mid \sigma_n.$$

wird verlangt, dass die Initialsymbolmengen aller Terme σ_i disjunkt sind.

$$\text{first}(\sigma_i) \cap \text{first}(\sigma_j) = O \quad \text{für alle } i \neq j$$

In der Syntax von Beispiel 3 gilt $x \in \text{first}(A)$ und $x \in \text{first}(B)$. Regel 1 ist also durch die Produktion $S = A \mid B$ verletzt. Nun ist es in diesem Beispiel allerdings leicht, diese Schwierigkeit zu umgehen, indem eine aequivalente Syntax definiert wird, welche Regel 1 beachtet:

$$S = C \mid xS.$$
$$C = y \mid z.$$

Leider genügt Regel 1 noch nicht, um alle Schwierigkeiten auszuschliessen, wie das folgenden Beispiel zeigt.

Beispiel 4

$$S = Ax.$$
$$A = [x].$$

Falls wir nun versuchen, den Satz x zu erkennen, geraten wir erneut in eine Sackgasse:

S	x
Ax	x
xx	x
x	ε

Die Schwierigkeit rührt von der Wahl der Produktion $A = x$ anstelle der

Produktion $A = \varepsilon$. Diese Situation kann nur entstehen, wenn ein Symbol die leere Folge erzeugen kann. Man beachte, dass dies stets der Fall ist, wenn eine Produktion die Form $A = \{s\}$ oder $A = [s]$ hat. Um die in Beispiel 4 charakterisierte Schwierigkeit auszuschliessen, adoptieren wir eine weitere einschränkende Regel.

Definition 2

Die Menge aller Symbole, die einer aus σ hergeleiteten Folge unmittelbar nachfolgen können, wird mit follow(σ) bezeichnet.

$$\text{follow}(\sigma) = \{s \mid S \dot{\to} \alpha\sigma' s\beta \wedge \sigma \dot{\to} \sigma'\}$$

Regel 2

Für jedes nicht-terminale Symbol A, aus welchen die leere Folge hergeleitet werden kann, muss die Menge seiner Initialsymbole disjunkt von der Menge der Folgesymbole sein.

$$A \dot{\to} \varepsilon \supset \text{first}(A) \cap \text{follow}(A) = O$$

In Beispiel 4 wird die Regel 2 durch das Symbol A verletzt, indem

$$\text{first}(A) = \text{follow}(A) = \{x\} \neq O$$

Man beachte auch, dass Regel 4 durch jede linksrekursive Produktion der Formen

$$A = A\xi \mid \ldots \quad \text{oder} \quad A = \{A\xi\} \quad \text{oder} \quad A = [A\xi]$$

verletzt wird, da $\text{first}(A) = \text{follow}(A) = \text{first}(\xi) \neq O$. Linksrekursive Produktionen sind daher durch die obigen Restriktionsregeln ausgeschlossen.

Aus den vorangehenden Ausführungen ist man vielleicht versucht, den Schluss zu ziehen, dass Schwierigkeiten mit syntaktischen Regeln stets dadurch zu lösen sind, dass eine aequivalente, den Regeln genügende Syntax konstruiert wird. Es ist daher angezeigt, an dieser Stelle daran zu erinnern, dass die Syntax nur Mittel zu einem höheren Zweck ist, nämlich zur Sichtbarmachung der Bedeutung eines Satzes. Wird nun eine Syntax umgeformt, so muss stets beachtet werden, dass damit die Bedeutungsstruktur der Sprache nicht in Mitleidenschaft gezogen wird. Die folgenden Produktionen definieren z.B. eine Form von Ausdrücken mit Operanden a, b, c, und dem Operator "-", den wir als Subtraktion auffassen wollen.

S = A | S-A.
A = a | b | c.

Da die linksrekursive Produktion für S Regel 1 verletzt, versuchen wir, eine aequivalente Syntax zu erstellen, und finden die Lösung als

S = AB.
B = [-S].
A = a | b | c.

Während nun aber die erste Version dem Satz a-b-c die Struktur zuweist, die durch Klammerung ((a-b)-c) hervorgehoben werden kann, so erhält der gleiche Ausdruck von der zweiten Version implizit die Struktur (a-(b-c)) zugewiesen. Unter Annahme der üblichen Regeln der Subtraktion ist es leicht ersichtlich, dass die beiden Versionen zwar syntaktisch, nicht aber semantisch aequivalent sind. Die Lehre, die wir aus dem Beispiel ziehen, ist, dass bei der Umformung einer Syntax stets auch die Bedeutungsstruktur der Sprache mitberücksichtigt werden muss.

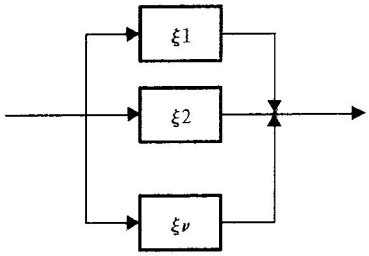
3. Syntax Graphen

Die Darstellung einer Syntax in BNF ist nur eine von verschiedenen Möglichkeiten. Eine andere, in vieler Hinsicht vorteilhafte Art der Darstellung beruht auf der Verwendung von Diagrammen oder Graphen. Der Hauptvorteil beruht dabei auf der besseren Ueberschaubarkeit. Wir schlagen nachfolgend eine Art der Graphen vor, die den Ablauf einer Satzerkennung im top-down Verfahren unmittelbar veranschaulicht. Wir geben ein einfaches Rezept, wie eine durch BNF definierte Syntax konsequent in entsprechende Graphen übersetzt werden kann. Der umgekehrte Vorgang ist natürlich ebenso leicht möglich. Wir nehmen an, dass die BNF bereits in dem Sinne normalisiert ist, dass jedes nicht-terminale Symbol durch eine einzige Produktionsfolge definiert ist. Die Uebersetzung ist dann durch nachfolgende Regeln bestimmt.

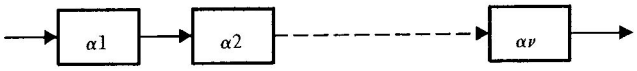
A1. Jedes nicht-terminale Symbol, definiert durch eine Produktionenfolge

$$A = \xi_1 \mid \xi_2 \mid \ldots \mid \xi_n.$$

wird als Graph mit der folgenden Struktur abgebildet:

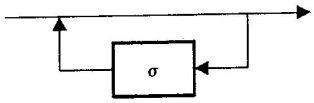


A2. Jeder Term $\alpha_1\ \alpha_2\ \ldots\ \alpha_n$ wird in einen Graphen der folgenden Struktur abgebildet:

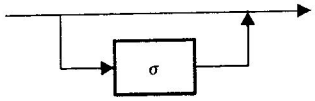


A3. Hat ein Element die Form $\{\sigma\}$, so wird es in die folgende Graphenstruktur

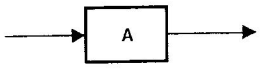
abgebildet:



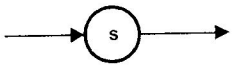
A4. Hat ein Element die Form [σ], so wird es in die folgende Struktur abgebildet:



A5. Ist ein Element ein nicht-terminales Symbol A, so wird es durch ein Rechteck eingefasst.



A6. Ist ein Element ein Terminalsymbol s, so wird es durch einen Kreis gekennzeichnet.



Beispiel 5

Wir adoptieren im nachfolgenden Beispiel die Konvention, dass alle terminalen Symbole durch Einklammerung mittels Gänsefüsschen eindeutig erkennbar gemacht werden. Diese Notation verwenden wir für alle späteren Beispiele.

```
A = "x" | "(" B ")".
B = AC.
C = {"+" A}.
```

Die Zeichen +, (,), und x sind also hier terminale Symbole der zu definierenden Sprache, während { und } als Metasymbole auftreten. Die

folgenden Sätze gehören z.B. zu dieser Sprache:

x
(x)
(x+x)
((x))
((x+(x+x)))

Die obigen Übersetzungsregeln verwandeln die drei Produktionen in die in Fig.1 abgebildeten Graphen. Indem C in B und B in A direkt substituiert werden, ergibt sich als Zusammenfassung der einzige, in Fig.2 dargestellte Graph.

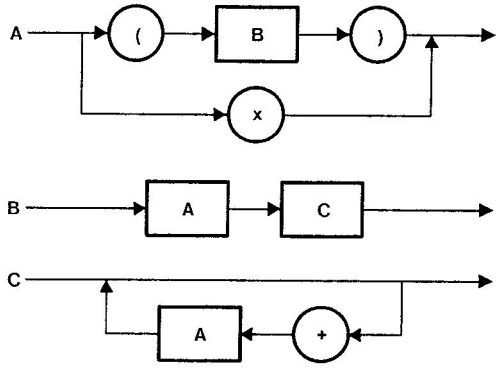


Fig. 1. Syntax nach Beispiel 5

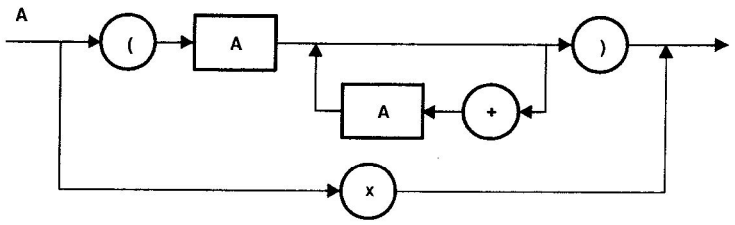


Fig. 2. Reduzierter Graph nach Beispiel 5

Die Graphen sind also eine Alternative zur BNF. Sie sind in Bezug auf Übersichtlichkeit der BNF vorzuziehen und vermitteln eine unmittelbarere Vorstellung über den Prozess der Satzerkennung. Es fragt sich daher, wie sich

die beiden postulierten Regeln im graphischen Bild äussern. In der Tat wird ihr Sinn dabei erst recht deutlich. Regel 1 besagt nämlich, dass bei jeder Verzweigung im Graphen der zu beschreitende Ast eindeutig aus der Kenntnis des nächsten Symbols hervorgehen muss. Dies impliziert, dass keine zwei Äste mit dem gleichen Symbol beginnen dürfen. Regel 2 verlangt, dass ein Graph, der ohne Einlesen eines Symbols traversiert werden kann, als "durchsichtig" betrachtet wird. Es muss eindeutig entscheidbar sein, ob das nächste Symbol zu dem Graphen oder bereits zu seinem Nachfolger gehört. Die Mengen der Anfangssymbole und der möglicherweise nachfolgenden Symbole müssen daher disjunkt sein.

Es ist nun sehr einfach zu verfizieren, ob eine Menge von Graphen diesen beiden Regeln genügt. Es ist dabei durchaus nicht notwendig, auf die Syntax in BNF zurückzugreifen. Im Gegenteil erweist sich die übersichtliche graphische Darstellung als besonders geeignet. Vorsorglicherweise bestimmen wir zuerst wiederum für jedes nicht-terminale Symbol A seine zugehörigen Mengen first(A) und follow(A). Wir nennen ein System von Graphen, das die beiden einschränkenden Regeln erfüllt, ein *deterministisches* Graphensystem.

4. Aufbau eines Parsers für eine gegebene Syntax

Falls eine Syntax durch einen deterministischen Graphen darstellbar ist, so lässt sich dieses Programm sehr systematisch aus dem Graphensystem herleiten. Die einzelnen Graphen entsprechen den zu erkennenden syntaktischen Kategorien und werden in einzelne Prozeduren abgebildet. Jeder Graph stellt sozusagen das Flussdiagramm der entsprechenden Prozedur dar. Die Übersetzung des Graphensystems in ein Programm lässt sich wiederum durch einzelne Regeln beschreiben, ganz analog zur Übertragung von BNF in graphische Form.

Die erhaltenen Prozeduren werden in ein Hauptprogramm eingebettet, das als Umgebung dient. Darin wird die Eingabefolge vereinbart, und ferner muss eine Prozedur gegeben sein, welche jeweils das nächste Symbol liefert. Der Einfachheit halber wollen wir vorerst annehmen, dass jedes Symbol aus einem einzelnen Zeichen besteht, wie sie in den üblichen Computer-Zeichensätzen vorkommen. Das Hauptprogramm wird also stets die folgende Form haben:

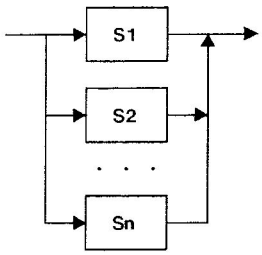
```
MODULE Parser;
  FROM Terminal IMPORT Read;
  VAR ch: CHAR;
(*Vereinbarungen der einzelnen Prozeduren*)
BEGIN Read(ch); S
END .
```

Dabei stellt S die dem Startsymbol S entsprechende Prozedur dar. Die einzelnen Übersetzungsschritte sind durch die folgenden Regeln B1 bis B6 bestimmt. Es ist dabei vorteilhaft, dass zuerst die Anzahl der nicht-terminalen Symbole reduziert wird, indem wenn möglich einzelne Graphen direkt in anderen Graphen eingesetzt werden, wie dies im obigen Beispiel 4 gezeigt wurde. Wir bezeichnen das Programm, das aus der Übersetzung eines Graphen S hervorgeht, mit P(S).

B1. Jede Struktur mit der Form

wird in eine bedingte oder selektive Anweisung übersetzt:

```
IF ch IN L1 THEN P(S1)
```

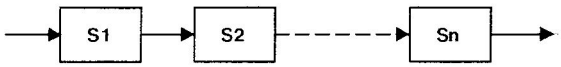



```
ELSIF ch IN L2 THEN P(S2)
.................
ELSIF ch IN Ln THEN P(Sn)
ELSE error
END

CASE ch OF
   L1: P(S1) |  L2: P(S2) |  ...   | Ln: P(Sn)
END
```

Dabei bezeichnet L_i die Menge first(S_i). (Programmierhinweis: Wenn L aus einem einzigen Symbol s besteht, so soll natürlich der Ausdruck ch IN L durch ch = s ersetzt werden).

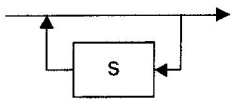
B2. Jede Struktur der Form



wird in eine Anweisungsfolge übersetzt:

```
P(S1); P(S2);  ...   ; P(Sn)
```

B3. Jede Struktur der Form

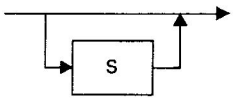


wird in eine repetitive Anweisung übersetzt:

```
WHILE ch IN L DO  P(S)  END
```

L stellt die Menge first(S) dar (siehe Hinweis unter B1).

B4. Jede Struktur der Form



wird in eine bedingte Anweisung übersetzt:

```
IF ch IN L THEN  P(S)  END
```

B5. Jede Referenz zu einem Graphen (rechteckige Figur) wird in einen Aufruf der dem Graphen entsprechenden Prozedur übersetzt.

B6. Jede Referenz zu einem Terminalsymbol x (runde Figur) wird wie folgt in eine Leseanweisung übersetzt:

```
IF ch = "x" THEN Read(ch) ELSE error END
```

Dabei stellt *error* eine nicht näher spezifizierte Prozedur dar, die aufgerufen wird, sobald eine inkorrekte Symbolfolge entdeckt wird.

Die Anwendung dieser Regeln soll nun am Beispiel 5 gezeigt werden. Sie resultiert im folgenden Programm 1. Dabei wurden allerdings bereits einige offensichtliche Vereinfachungen vorgenommen.

Programm 1: Parser für Sprache aus Beispiel 5

```
MODULE Parser;
  FROM Terminal IMPORT Read;
  VAR ch: CHAR;

  PROCEDURE A;
  BEGIN
    IF ch = "x" THEN Read(ch)
    ELSIF ch = "(" THEN
      Read(ch); A;
      WHILE ch = "+" DO Read(ch); A END ;
      IF ch = ")" THEN Read(ch) ELSE error END
    ELSE error
    END
```

```
  END A;

BEGIN Read(ch); A
END Parser.
```

Programm 1: Parser für Sprache aus Beispiel 4

Eine buchstabengetreue Übersetzung hätte z.B. zu folgender Anweisung geführt.

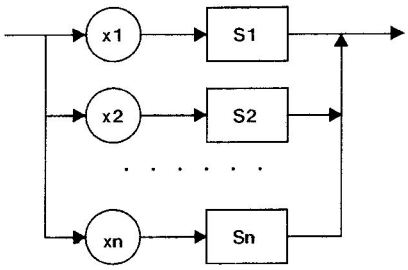
```
IF ch = "x" THEN
  IF ch = "x" THEN Read(ch) ELSE error END
ELSE  ...
END
```

welche offensichtlich in die Anweisung

```
IF ch = "x" THEN Read(ch) ELSE  ...  END
```

vereinfacht werden kann. Solche Vereinfachungen sind tatsächlich relativ oft möglich. Zwei häufig vorkommende Situationen führen zu den folgenden Zusatzregeln:

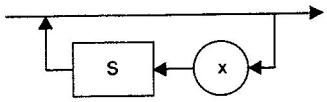
B1a. Eine Struktur der Form



wird in folgende Anweisung übersetzt:

```
IF ch = "x1" THEN read(ch); P(S1)
ELSIF ch = "x2" THEN read(ch); P(S2)
   . . . . . .
ELSIF ch = "xn" THEN read(ch); P(Sn)
ELSE error
END
```

B2a. Eine Struktur der Form



wird in die folgende while-Anweisung übersetzt:

```
WHILE ch = "x" DO Read(ch); P(S) END
```

Die Prozedur *error* bleibt vorläufig unbestimmt. Solange wir lediglich daran interessiert sind, herauszufinden, ob eine gegebene Zeichenfolge syntaktisch legitim sei oder nicht, genügt es, wenn beim Aufruf von *error* der Prozess abgebrochen wird. In der Praxis ist dies natürlich nicht der Fall, da normalerweise erwartet wird, dass der nachfolgende Text weiter untersucht werde. Die daraus hervorgehenden Probleme und Techniken werden im Kapitel 9 behandelt.

5. Tabellen-gesteuerte Syntax Analyse

Im vorangehenden Kapitel wurde eine Methodik beschrieben, die es erlaubt, für irgend eine Syntax den zugehörigen Parser zu programmieren, sofern die Syntax den zwei einschränkenden Regeln genügt. Da jedoch immer dasselbe Prinzip der Analyse verwendet wird, ist es auch möglich, ein einziges, allgemeines Programm zu erstellen, das für alle entsprechenden Sprachen geeignet ist. Dieses allgemeine Programm wird je nach Sprache mit Tabellen versehen, welche deren Syntax in codierter Form darstellen. Der Parser wird dann sozusagen von diesen - während der Analyse unveränderten - Tabellen gesteuert. Er verkörpert den Algorithmus, nach dem wir einen Satz erkennen, wenn wir anhand der Syntax-Graphen vorgehen.

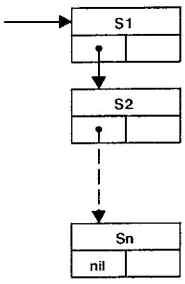
Die Syntax muss also in einer Form dargestellt werden, wie sie einem Computer Programm zugänglich ist. Weder BNF noch Graphen sind dafür sehr geeignet. Es ist naheliegend, von den Graphen als Fluss-Diagramme auszugehen, und diese in eine geeignete Daten-Struktur zu verwandeln. Dabei wird kaum eine einfache Tabellenform (Array) in Frage kommen, sondern viel eher eine Listenstruktur ohne festes, vorgegebenes Format. Den einzelnen Symbolen entsprechen Daten-Elemente (Knoten), während die Pfade (Kanten) durch Zeiger (pointers) dargestellt werden.

In der Programmiersprache Modula-2 ist der Record die geeignete Struktur, um Knoten in Listen darzustellen. Im vorliegenden Fall werden in jedem Datenelement vier Felder benötigt. Das erste Feld stellt das Symbol dar, welches dem Knoten entspricht. Zwei weitere Felder enthalten Zeiger, die entweder auf den nachfolgenden Knoten oder die Alternative verweisen, falls diese existieren. Ein weiteres Feld dient zur Angabe, ob es sich um ein terminales oder ein nicht-terminales Symbol handelt. Diese Record-Struktur wird vorteilhafterweise als Record-Varianten-Typ definiert.

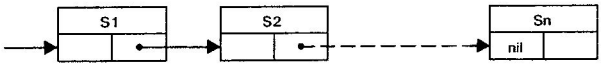
```
TYPE pointer = POINTER TO node;
     node = RECORD suc, alt: pointer;
              CASE terminal: BOOLEAN OF
                TRUE:  (tsym: char) |
                FALSE: (nsym: hpointer)
              END
            END
```

Es liegt nun auf der Hand, die Uebersetzung eines Systems von Graphen in eine entsprechende Datenstruktur wiederum durch eine Liste von festen Regeln zu definieren. Danach ist es möglich, sogar ein Programm zu konstruieren, welches diese Umformung automatisiert. Die Regeln sind in der Tat den bisher angegebenen Regeln sehr ähnlich.

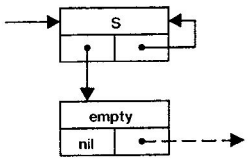
C1. Eine Graphenstruktur, wie sie unter B1 angegeben ist, wird in folgende Listenstruktur abgebildet:



C2. Eine Graphenstruktur, wie sie unter B2 angegeben ist, wird in folgende Listenstruktur abgebildet:

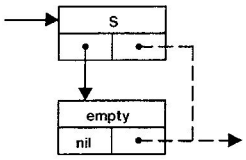


C3. Eine Schleifenstruktur (siehe B3) wird wie folgt abgebildet:



C4. Die unter B4 angegebene Struktur wird wie folgt dargestellt:

Zur Veranschaulichung verwenden wir nochmals Beispiel 5 (siehe Fig.2). Nach



obigen Regeln wird dieser Graph in folgende Datenstruktur umgewandelt.

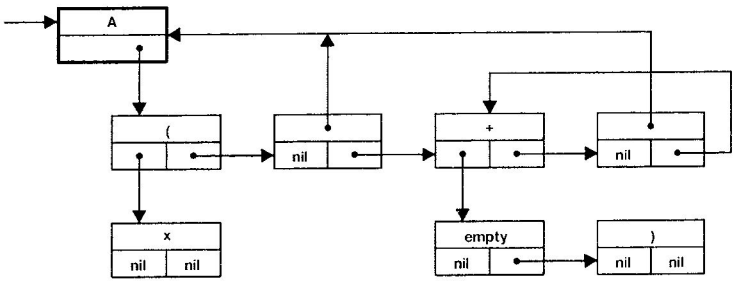


Fig. 3. Datenstruktur entsprechend dem Syntaxgraphen der Fig.2

Die obige Datenstruktur wird durch einen sogenannten Leitknoten identifiziert. Er enthält den Namen des entsprechenden nicht-terminalen Symbols in druckbarer Form. Diese Information ist für die Satzzerlegung im engeren Sinne nicht nötig. Sie wird aber im später zu beschreibenden Algorithmus verwendet, um den Ablauf der Zerlegung sichtbar zu machen. Der Datentyp *hpointer* wird dabei wie folgt vereinbart:

```
TYPE hpointer = POINTER TO header;

TYPE header = RECORD symbol: CHAR;
                     entry: pointer
              END
```

Der zu entwerfende Algorithmus zur Satzzerlegung (Parser) besteht nun aus einer repetierten Anweisung, die besagt, wie von einem Knoten der Datenstruktur zum nächsten übergegangen wird. Das Programm beschreibt also, wie ein Graph, dargestellt als Listenstruktur, interpretiert wird. Falls ein Knoten erreicht wird, der ein nicht-terminales Symbol darstellt, so wird derselbe Algorithmus auf die dem Symbol entsprechende Datenstruktur angewendet, bevor in der gegenwärtig vorliegenden Struktur weitergeschritten wird. Die

Prozedur *parse*, die das Traversieren einer Struktur beschreibt, wird also rekursiv aufgerufen. Falls ein Koten ein Terminalsymbol darstellt, so wird geprüft, ob das nächste Eingabesymbol (*sym*) damit identisch ist. Ist dies der Fall, so wird dieses Symbol (Zeichen) eingelesen und zum Nachfolger übergegangen (*suc*). Ansonst wird die Alternative (*alt*) inspiziert, falls eine existiert.

```
PROCEDURE parse(goal: hpointer; VAR match: BOOLEAN);
   VAR s: pointer;
BEGIN s := goal↑.entry;
   REPEAT
      IF s↑.terminal THEN
         IF s↑.tsym = sym THEN
            match := TRUE; getsym
         ELSE match := (s↑.sym = empty)
         END
      ELSE parse(s↑.nsym, match)
      END ;
      IF match THEN s := s↑.suc ELSE s := s↑.alt END
   UNTIL s = NIL
END parse
```

Das obige Programm besitzt die Eigenschaft, ein neues Ziel zu verfolgen, sobald ein solches durch ein vorliegendes nicht-terminales Symbol A angezeigt wird. Und zwar geschieht dies, ohne dass zuerst geprüft wird, ob das nächste Eingabesymbol zur Menge der Anfangssymbole first(A) gehöre. Dies bedingt aber, dass für jede Verzweigung in einem Syntax-Graphen gelten muss, dass höchstens ein einziger Ast mit einem nicht-terminalen Symbol beginnen darf. Falls ein Ast der leeren Folge entspricht, so darf sogar kein paralleler Ast mit einem nicht-terminalen Symbol beginnen. Wir formulieren diese Bedingungen nachfolgend als Einschränkungen der Form von Produktionen, und nehmen an, dass jedes nicht-terminale Symbol A durch eine einzige zusammengesetzte Produktion definiert ist. Diese muss eine der folgenden Formen haben:

1. $A = s_1\xi_1 \mid s_2\xi_2 \mid \ldots \mid s_n\xi_n$
   $A \in N,\ s_i \in T,\ \xi_i \in (N \cup T)^*,\ i \neq j \supset s_i \neq s_j$

2. $A = s_1\xi_1 \mid s_2\xi_2 \mid \ldots \mid s_n\xi_n \mid B\eta$
   $B \in N,\ h \in (N \cup T)^*,\ s_i \notin \text{first}(B)$ für $i = 1 \ldots n$

3. $A = s_1 c_1 \mid s_2 \xi_2 \mid \ldots \mid s_n \xi_n \mid \varepsilon$
   $s_i \notin \text{follow}(A)$ für $i = 1 \ldots n$

Weniger restriktive Regeln bedingen raffiniertere Algorithmen, die sich mit unterschiedlichem Aufwand realisieren lassen. Sogar von der "no backtracking" Regel kann abgewichen werden, indem das obige Programm entsprechend verallgemeinert wird. Dies bringt jedoch einen ins Gewicht fallenden Verlust an Effizienz mit sich, der für praktische Anwendungen möglichst vermieden werden sollte.

Die Darstellung einer Syntax mit Graphen hat neben Vorteilen einen entschiedenen Nachteil: sie kann nicht direkt von einem Computer eingelesen werden. In dieser Hinsicht erscheint die BNF als ideal, weil sie eine Syntax in konventioneller Form als Formeltext darstellt, der aus einzelnen Buchstaben und Zeichen besteht. Im folgenden Kapitel wird daher die Frage behandelt, wie Parser-Tabellen direkt aus einer BNF-Spezifikation erhalten werden.

6. Die Übersetzung von BNF-Produktionen in Tabellen

Im Unterschied zu den vorangegangenen Beschreibungen von Darstellungs-Transformationen streben wir in diesem Kapitel nicht eine informelle Definition, sondern ein formales Programm an. Dieses Ziel erscheint nicht nur deshalb interessant, weil sich damit die Umformung durch Computer automatisch und fehlerfrei durchführen lässt, sondern ganz besonders weil dieses Transformations-Programm als unmittelbare Anwendung der in Kapitel 4 beschriebenen Technik erscheint.

Dieser Umstand erklärt sich aus der Tatsache, dass die BNF - oder genauer die hier verwendete erweiterte BNF - dazu verwendet werden kann, sich selbst zu definieren. Wir bezeichnen diese nachfolgend genauer zu definierende Metasprache als Erweiterte Backus-Naur Form (EBNF). Das Ziel ist also, einen Parser für diese Metasprache zu konstruieren. Wir gehen dabei in folgenden vier Schritten vor:

Schritt 1: Definition der Syntax der EBNF, ausgedrückt in EBNF selbst.

Schritt 2: Konstruktion eines Parsers für diese Syntax nach den Regeln B1-B6.

Schritt 3: Ausbau dieses Parsers in einen Übersetzer, der Parser-Tabellen erzeugt.

Schritt 4: Integration dieses Übersetzers mit dem Tabellen-gesteuerten Parser.

Schritt 1.

In Übereinstimmung mit bisher befolgten Notationsregeln soll der Eingabetext mit den in EBNF formulierten Produktionen folgende Bedingungen erfüllen.

1. Nicht-terminale Symbole bestehen aus einem einzigen Grossbuchstaben.

2. Terminalsymbole sind einzelne Zeichen (nicht notwendigerweise Buchstaben), gekennzeichnet durch Anführungszeichen (literal).

Die Syntax, die eine EBNF-Produktion beschreibt, ergibt sich sodann aus folgenden Produktionen.

```
production =   identifier "=" expression "." .
expression =   term {"|" term}.
term =         factor {factor}.
factor =       letter | literal | "(" expression ")" | "[" expression "]" |
               "{" expression "}".
```

Schritt 2

Obwohl die obigen Regeln der lexikographischen Textdarstellung den Aufbau des Parsers im wesentlichen einfach gestalten, lässt sich doch das Einlesen des nächsten Symbols nicht direkt durch die Anweisung *Read(ch)* darstellen. Zumindest sollten doch Leerzeichen in beliebiger Anzahl zwischen den Symbolen eingestreut werden dürfen. Für das Einlesen postulieren wir daher eine Prozedur *GetSym*, welche der Variablen *sym* das nächste Symbol zuweist, ungeachtet möglicherweise eingestreuter Leerzeichen. Diese Prozedur stellt ein triviales Beispiel eines Mechanismus dar, der im allgemeinen ein *Scanner* genannt wird. Sein Zweck ist das Herausgreifen des nächsten Symbols nach den Regeln der gewählten lexikographischen Darstellung. So ist es z.B. üblich, dass gewisse Symbole aus mehreren Zeichen bestehen. Der Scanner bildet sodann Zeichenfolgen in Symbole ab. Wir erkennen darin den Unterschied zwischen Zeichen und Symbolen: Zeichen sind die atomaren Einheiten im Eingabetext des Computers, Symbole sind die Grundsteine der durch die Syntax unabhängig von bestimmten Zeichensätzen definierten Sprache.

Bevor wir zur Konstruktion eines Parsers schreiten, ist zu verifizieren, ob die vorliegende Syntax den einschränkenden Regeln 1 und 2 genügt. Danach wenden wir "mechanisch" die Übertragungsregeln B1 - B6 an und erhalten dabei das folgende Programm 2.

```
MODULE EBNF;
  FROM Terminal IMPORT Read, Write, WriteLn;
  CONST empty = " ";
  VAR sym: CHAR;

PROCEDURE GetSym;
BEGIN
   REPEAT Read(sym); Write(sym) UNTIL sym # " "
END GetSym;

PROCEDURE error;
```

```
BEGIN WriteLn;
   Write(" Syntax incorrect"); HALT
END error;

PROCEDURE expression;

  PROCEDURE term;

    PROCEDURE factor;
    BEGIN
      IF ("A" <= sym) & (sym <= "Z") THEN
        (*nonterminal symbol*) GetSym
      ELSIF sym = '"' THEN
        (*terminal symbol*) GetSym; GetSym;
        IF sym = '"' THEN GetSym ELSE error END
      ELSIF sym = "(" THEN
        GetSym; expression;
        IF sym = ")" THEN GetSym ELSE error END
      ELSIF sym = "[" THEN
        GetSym; expression;
        IF sym = "]" THEN GetSym ELSE error END
      ELSIF sym = "{" THEN
        GetSym; expression;
        IF sym = "}" THEN GetSym ELSE error END
      ELSE error
      END
    END factor;

  BEGIN factor;
    WHILE ("A" <= sym) & (sym <= "Z") OR (sym = '"') OR
      (sym = "(") OR (sym = "[") OR (sym = "{") DO factor
    END
  END term;

BEGIN term;
  WHILE sym = "|" DO GetSym; term END
END expression;

BEGIN (*main program*) GetSym;
  WHILE sym # "." DO
```

```
    IF ("A" <= sym) & (sym <= "Z") THEN GetSym ELSE error END;
    IF sym = "=" THEN GetSym ELSE error END ;
    expression;
    IF sym = "." THEN GetSym ELSE error END ;
    WriteLn
  END
END EBNF.
```

Programm 2: Syntax Analysis für EBNF

Schritt 3

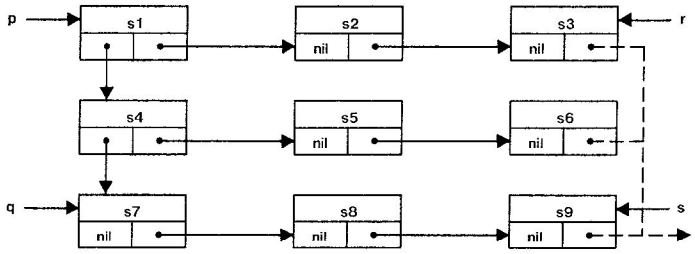
Das nächste Ziel ist die Erweiterung des erhaltenen Programms in einen Übersetzer, der die gewünschte Datenstruktur erzeugt. Leider lässt sich aber dieser Schritt nicht in einer Art formalisieren oder systematisieren wie der letzte. Mangels formaler Aufbauregeln greifen wir zum naheliegenden Rezept, die jeder syntaktischen Einheit entsprechende Struktur nochmals aufzuzeichnen, und die Parameter der sie erzeugenden Prozedur genau zu definieren. Jede aufgerufene Prozedur erzeugt also eine dem von ihr eingelesenen Text entsprechende Datenstruktur. Diese wird sodann von der aufrufenden Prozedur als Baustein in der von ihr selbst aufgebauten Datenstruktur eingefügt. Dieser Vorgang ist natürlich im allgemeinen rekursiv. Da es sich bei den erzeugten Produkten um durch Zeiger verkettete Strukturen handelt, liegt es auf der Hand, als Resultate nicht die Strukturen selbst, sondern ihre Zeiger zu verwenden. Jede der drei Generierungs- Prozeduren *factor, term* und *expression* erhält vier Parameter, die mit p, q, r und s bezeichnet sind. Ihre Bedeutung ist aus der folgenden Datenstruktur ersichtlich, die dem Syntax-Ausdruck

· s1 s2 s3 | s4 s5 s6 | s7 s8 s9

entspricht. Die vier Zeiger sind das Resultat der Generierung durch die Prozedur *expression.*

Der Hauptzweck der Prozedur *factor* ist das Erzeugen von neuen Elementen (Knoten) in der Datenstruktur, die den gelesenen Symbolen entsprechen. Die beiden Prozeduren *term* und *expression* hingegen haben die Aufgabe, diese Elemente miteinander zu verketten. Dabei verwendet *term* das Zeigerfeld *suc,* und *expression* benützt das Feld *alt.* Die Einzelheiten sind aus Programm 3 ersichtlich.

Es sei hier noch auf eine Einzelheit bei der Verarbeitung von nicht-terminalen



Symbolen hingewiesen. Möglicherweise kann ein solches Symbol innerhalb eines Faktors vorkommen, bevor es als Linksteil einer Produktion auftritt und dabei definiert wird. Es ist daher notwendig festzuhalten, ob ein vorgekommenes Symbol bereits definiert ist. Dazu werden alle nicht-terminalen Symbole in Records vom Typ *header* registriert, und das Feld *entry* gibt an, ob eine entsprechende Datenstruktur bereits aufgebaut wurde. Diese Records werden in einer Liste linear verkettet. Die Variable *list* stellt den Anfang, *sentinel* das Ende dieser Kette dar, wobei das Ende ein fiktives Element mit dem Zweck ist, den Suchvorgang zu vereinfachen. Dieser Suchvorgang wird von der Hilfsprozedur *find(sym,h)* verkörpert. Wird das Symbol *sym* gefunden, so wird der Zeiger zum entsprechenden Element der Variablen *h* zugewiesen. Ist das Symbol nicht in der Liste vorhanden, so wird ein Element angefügt. Weitere Einzelheiten sind dem ersten Teil von Programm 3 zu entnehmen.

Schritt 4

Der Übersetzer wird nun mit dem allgemeinen Parser aus Abschnitt 5 vereinigt. Sowohl das Programm wie die Eingabedaten bestehen aus drei Teilen. Zuerst folgt der Parser für EBNF, respektive die Syntax der zu behandelnden Sprache ausgedrückt durch EBNF Produktionen. Dann folgt eine Anweisung, die das Startsymbol einliest. Schliesslich folgen der allgemeine Parser, beziehungsweise die der vorher definierten Sprache angehörigen und zu erkennenden Sätze. (Das Programm erwartet, dass das Startsymbol durch ein vorangehendes $-Zeichen gekennzeichnet wird.)

```
MODULE GeneralParser;
  FROM SYSTEM IMPORT TSIZE;
  FROM InOut IMPORT OpenInput, OpenOutput, Done, Read,
    Write, WriteLn, WriteString, CloseInput, CloseOutput;
  FROM Heap IMPORT ALLOCATE;
```

```
CONST empty = " "; ESC = 33C;

TYPE NodePtr = POINTER TO Node;
   HeadPtr = POINTER TO Header;

   Node = RECORD suc, alt: NodePtr;
            CASE terminal: BOOLEAN OF
              TRUE:  tsym: CHAR |
              FALSE: nsym: HeadPtr
            END
          END ;

   Header = RECORD sym: CHAR;
               entry: NodePtr;
               suc: HeadPtr
            END ;

VAR list, sentinel, h: HeadPtr;
    q,r,s: NodePtr;
    sym: CHAR;
    noerr: BOOLEAN;

PROCEDURE GetSym;
BEGIN
   REPEAT Read(sym); Write(sym)
   UNTIL (sym > " ") OR (sym = ESC)
END GetSym;

PROCEDURE find(s: CHAR; VAR h: HeadPtr);
  VAR h1: HeadPtr;
BEGIN h1 := list; sentinel↑.sym := s;
   WHILE h1↑.sym # s DO h1 := h1↑.suc END ;
   IF h1 = sentinel THEN (*insert*)
      ALLOCATE(sentinel, TSIZE(Header));
      h1↑.suc := sentinel; h1↑.entry := NIL
   END ;
   h := h1
END find;

PROCEDURE error;
BEGIN WriteLn;
```

```
  WriteString(" incorrect syntax "); noerr := FALSE
END error;

PROCEDURE link(p,q: NodePtr);
   VAR t: NodePtr;
BEGIN (*insert q in places indicated by linked chain p*)
   WHILE p # NIL DO
      t := p; p := t↑.suc; t↑.suc := q
   END
END link;

PROCEDURE expression(VAR p,q,r,s: NodePtr);
   VAR q1, s1: NodePtr;

PROCEDURE term(VAR p,q,r,s: NodePtr);
   VAR p1, q1, r1, s1: NodePtr;

   PROCEDURE factor(VAR p,q,r,s: NodePtr);
      VAR a: NodePtr;  h: HeadPtr;
   BEGIN
      IF ("A" <= sym) & (sym <= "Z") THEN
         ALLOCATE(a, TSIZE(Node)); find(sym,h);
         WITH a↑ DO
            terminal := FALSE; nsym := h; alt := NIL;
            suc := NIL
         END ;
         p := a; q := a; r := a; s := a; GetSym
      ELSIF sym = '"' THEN (*terminal symbol*)
         GetSym; ALLOCATE(a, TSIZE(Node));
         WITH a↑ DO
            terminal := TRUE; tsym := sym; alt := NIL;
            suc := NIL
         END ;
         p := a; q := a; r := a; s := a; GetSym;
         IF sym = '"' THEN GetSym ELSE error END
      ELSIF sym = "(" THEN
         GetSym; expression(p,q,r,s);
         IF sym = ")" THEN GetSym ELSE error END
      ELSIF sym = "[" THEN
         GetSym; expression(p,q,r,s);
```

```
         ALLOCATE(a, TSIZE(Node));
         WITH a↑ DO
            terminal := TRUE; tsym := empty; alt := NIL;
            suc := NIL
         END ;
         q↑.alt := a; s↑.suc := a; q := a; s := a;
         IF sym = "]" THEN GetSym ELSE error END
      ELSIF sym = "{" THEN
         GetSym; expression(p,q,r,s); link(r,p);
         ALLOCATE(a, TSIZE(Node));
         WITH a↑ DO
            terminal := TRUE; tsym := empty; alt := NIL;
            suc := NIL
         END ;
         q↑.alt := a; q := a; r := a; s := a;
         IF sym = "}" THEN GetSym ELSE error END
      ELSE error
      END
   END factor;

BEGIN (*term*)  factor(p,q,r,s);
   WHILE ("A" <= CAP(sym)) & (CAP(sym) <= "Z") OR
         (sym = '"') OR (sym = "(" ) OR (sym = "[" ) OR
         (sym = "{" ) DO
      factor(p1,q1,r1,s1); link(r,p1); r := r1; s := s1
   END
END term;

BEGIN (*expression*)  term(p,q,r,s);
   WHILE sym = "|" DO
      GetSym; term(q↑.alt, q1, s↑.suc, s1); q := q1; s := s1
   END
END expression;

PROCEDURE parse(goal: HeadPtr; VAR match: BOOLEAN);
   VAR s: NodePtr;
BEGIN s := goal↑.entry;
   REPEAT
      IF s↑.terminal THEN
         IF s↑.tsym = sym THEN
```

```
            match := TRUE; GetSym
          ELSE match := (s↑.tsym = empty)
          END
        ELSE parse(s↑.nsym, match)
        END ;
        IF match THEN s := s↑.suc ELSE s := s↑.alt END
    UNTIL s = NIL
END parse;

BEGIN (*main program*) OpenInput("EBNF");
  IF Done THEN noerr := TRUE;
    ALLOCATE(sentinel, TSIZE(Header)); list := sentinel;

    (*read productions and build data structure*)
    LOOP GetSym;
      IF sym = "$" THEN EXIT END ;
      find(sym,h);
      GetSym; IF sym = "=" THEN GetSym ELSE error END ;
      expression(h↑.entry, q,r,s); link(r, NIL);
      IF sym # "." THEN error END
    END ;

    IF noerr THEN h := list;
      (*check whether all symbols are defined*)
      WHILE h # sentinel DO
        IF h↑.entry = NIL THEN
          WriteString(" undefined symbol "); Write(h↑.sym);
          WriteLn; noerr := FALSE
        END ;
        h := h↑.suc
      END
    END ;

    IF noerr THEN
      (*read goal symbol*)
      GetSym; find(sym,h); WriteLn; CloseInput;
      (*read and parse sentences*)
      LOOP Write(">"); GetSym;
        IF sym = ESC THEN EXIT END ;
        parse(h, noerr);
```

```
        IF noerr & (sym = ESC) THEN WriteString("  correct")
        ELSE WriteString("  incorrect")
        END ;
        WriteLn
      END
    ELSE CloseInput
    END ;
    WriteLn
  END
END GeneralParser.
```

Programm 3: Allgemeiner Parser

Es ist bemerkenswert, dass dieses Programm einzig durch Einfügen von zusätzlichen Anweisungen in ein bestehendes Programm erzeugt wurde. Das bestehende Programm befasste sich lediglich mit der Erkennung von Sätzen, und kann als Rahmen für ein neues Programm betrachtet werden, das Sätze nicht nur erkennt, sondern in Datenstrukturen übersetzt. Diese Methode der Programm-Entwicklung ist sehr empfehlenswert. Sie gestattet nämlich die Konzentration auf einen bestimmten Aspekt (z.B. die Satzerkennung), bevor andere Aspekte (z.B. die Erzeugung von Datenstrukturen) ebenfalls berücksichtigt werden müssen. Dadurch wird die Arbeit des systematischen Aufbaus und der Verifikation bedeutend erleichtert. Zumindest hilft die Methode dem Programmierer, sich besser und vor allem fortlaufend von der Richtigkeit des komplexen Programmes überzeugen zu können. Die Methode wird *schrittweise Verfeinerung* (stepwise refinement) oder schrittweiser Ausbau genannt. In unserem eher einfachen Beispiel dieser Technik besteht der Ausbau lediglich aus zwei Schritten. Kompliziertere Sprachen und komplexere Übersetzungsaufgaben verlangen oft eine wesentlich grössere Zahl solcher Schritte. Eine sehr ähnliche Entwicklung in drei Schritten wird in Kapiteln 8 bis 11 behandelt.

Programm 3 zeigt deutlich, dass der Tabellen-gesteuerte - oder eher Datenstruktur-gesteuerte - Algorithmus der Satzzerlegung einen Grad von Flexibilität und Einfachheit aufweist, wie er dem direkt programmierten Parser fehlt. Diese zusätzliche Flexibilität ist, obwohl sie im allgemeinen nicht benötigt wird, der eigentliche Kern der Idee der erweiterbaren Sprachen (extensible languages). Dies sind Sprachen, beziehungsweise Übersetzer, die sich durch Hinzufügen von weiteren Syntaxregeln leicht erweitern lassen. Der Programmierer kann also nach Belieben für jedes Programm

Spracherweiterungen angeben, und zwar in Form von Produktionen, welche die Übersetzer-Tabellen vorübergehend vergrössern. Ein noch ehrgeizigerer Plan lässt es sogar zu, dass die Sprache für lokale Bereiche innerhalb eines Programmes beliebig erweiterbar (und einschränkbar) ist. Dies wird möglich, wenn Programm 3 derart abgeändert wird, dass sich Abschnitte von Produktionen und Sätzen (Programmteilen) in der Eingabe abwechseln lassen. Ob sich eine solche Flexibilität in der Praxis als wünschbar oder sogar nützlich erweist, ist allerdings mehr als fraglich. Entwicklungen von solchen erweiterbaren Sprachen und deren Übersetzer sind denn auch bisher von wenig Erfolg belohnt worden. Der Hauptgrund liegt wohl darin, dass es ganz wesentlich schwieriger ist, die Bedeutung der angefügten Erweiterungen zu erklären als deren Struktur. Dieser Teil der Aufgabe hat bisher jedem Versuch einer leicht verständlichen Formalisierung widerstanden. Abgesehen davon liegt der Zweck von Sprachen in der gegenseitigen Verständigung. Dazu sind immer ein Konsensus und Konventionen notwendig. Die beliebige Erfindbarkeit von eigenen Sprachen läuft aber diesem Endziel offensichtlich zuwider. Wir ziehen daraus die Konsequenzen und konzentrieren uns im Rest dieses Kapitels ausschliesslich auf die Entwicklung eines einfachen Übersetzers für eine einzige, vorgegebene, kleine Programmiersprache.

7. Die Programmiersprache PL/0.

Dieses und die folgenden Kapitel sind der Entwicklung einer elementaren Programmiersprache und ihres Compilers gewidmet. Sie heisse PL/0. Einerseits ist es unumgänglich, diese Sprache und den Compiler einfach zu halten, damit sie in den Rahmen dieses Buches passen. Anderseits liegt der Wunsch nahe, die wichtigsten Grundprinzipien von Programmiersprachen und Compilations-Techniken darlegen zu können. Daraus ergeben sich bereits die Randbedingungen für die Art der zu wählenden Sprache PL/0. Ohne Zweifel könnte sie auch einfacher oder komplizierter Gewählt werden. PL/0 stellt lediglich einen der möglichen Kompromisse zwischen genügender Einfachheit für eine übersichtliche Darstellung und genügender Komplexität dar, die das ganze Projekt realistisch und lohnend macht.

Was Programmstrukturen betrifft, so ist PL/0 verhältnismässig gut ausgebaut. Als elementare Anweisung finden wir die Wertzuweisung. Die zusammengesetzten Anweisungen verkörpern die Konzepte der Anweisungsfolge, der bedingten und der repetierten Anweisung, letztere in der Form von konventionellen if- und while-Strukturen. PL/0 enthält ferner das wichtige Konzept des Unterprogramms, dargestellt durch Prozedur-Vereinbarungen und Aufrufe.

Im Bereich der Datentypen und Datenstrukturen hingegen ist PL/0 äusserst frugal. Der einzige Typ ist die ganze Zahl (integer), und Datenstrukturen fehlen sogar gänzlich. Es ist also möglich, ganzzahlige Konstanten und Variablen zu vereinbaren, sowie Ausdrücke mit den arithmetischen Grundoperationen zu bilden.

Prozeduren stellen logische Einheiten von Operationen dar. Es liegt daher nahe, das wichtige Konzept der Lokalität von Objekten (Konstanten, Variablen, Prozeduren) zu verbinden. PL/0 bietet die Möglichkeit, Objekte als lokal zu vereinbaren, d.h. so, dass sie nur innerhalb der Prozedur Gültigkeit haben, in der sie vereinbart sind.

Ferner enthält PL/0 einfache Anweisungen zur Ein- und Ausgabe von Daten. Das Einlesen und Zuweisen eines Wertes an eine Variable v wird kurz mit ?v angezeigt, und die Ausgabe eines Wertes x mit !x.

Die obige, kurze Charakterisierung der Sprache hat den Hauptzweck, die nötige Intuition zu vermitteln, um die nachfolgend definierte Syntax zu erfassen und den mit den einzelnen Strukturen verknüpften Sinn zu begreifen. Die Syntax ist nachfolgend in EBNF und am Ende dieses Kapitels mittels Graphen dargestellt.

PL/0 Syntax:

```
program   =   block "." .
block     =   [ "CONST" ident "=" number {"," ident "=" number} ";"]
              [ "VAR" ident {"," ident} ";"]
              { "PROCEDURE" ident ";" block ";"} statement .
statement =   [ ident ":=" expression | "CALL" ident |
              "?" ident | "!" expression |
              "BEGIN" statement {";" statement} "END" |
              "IF" condition "THEN" statement |
              "WHILE" condition "DO" statement ] .
condition =   "ODD" expression |
              expression ("="|"#"|"<"|"<="|">"|">=") expression .
expression =  ["+"|"-"] term {("+"|"-") term}.
term      =   factor {("*"|"/") factor}.
factor    =   ident | number | "(" expression ")" .
```

Ein einziges Beispiel eines PL/0 Programmes möge genügen, um die Elemente der Sprache vorzuführen und so ein Bild der Sprache selbst zu vermitteln. Das gewählte Programm enthält Prozeduren zur Multiplikation, Division und Berechnung des grössten gemeinsamen Teilers zweier natürlicher Zahlen, sowie zur Berechnung der Fakultät mittels Rekursion.

```
VAR x,y,z, q,r, n,f;

PROCEDURE multiply;
   VAR a,b;
BEGIN a := x; b := y; z := 0;
   WHILE b > 0 DO
   BEGIN IF ODD z THEN z := z + a;
        a := 2*a; b := b/2
   END
END ;
```

```
PROCEDURE divide;
   VAR w;
BEGIN r := x; q := 0; w := y;
   WHILE w <= r do  w := 2*w;
   WHILE w > y do
   BEGIN q := 2*q; w := w/2;
      IF w <= r THEN
         BEGIN r := r - w; q := q + 1
         END
   END
END ;

PROCEDURE gcd;
   VAR f,g;
BEGIN f := x; g := y;
   WHILE f # g DO
      BEGIN IF f < g THEN g := g - f;
            IF g < f THEN f := f - g;
      END ;
   z := f
END ;

PROCEDURE fact;
BEGIN
  IF n > 1 THEN
    BEGIN f := n*f; n := n-1; CALL fact
    END
END ;

BEGIN
   ?x; ?y; CALL multiply; !z;
   ?x; ?y; CALL divide; !q; !r;
   ?x; ?y; CALL gcd; !z;
   ?n; f := 1; CALL fact; !f
END .
```

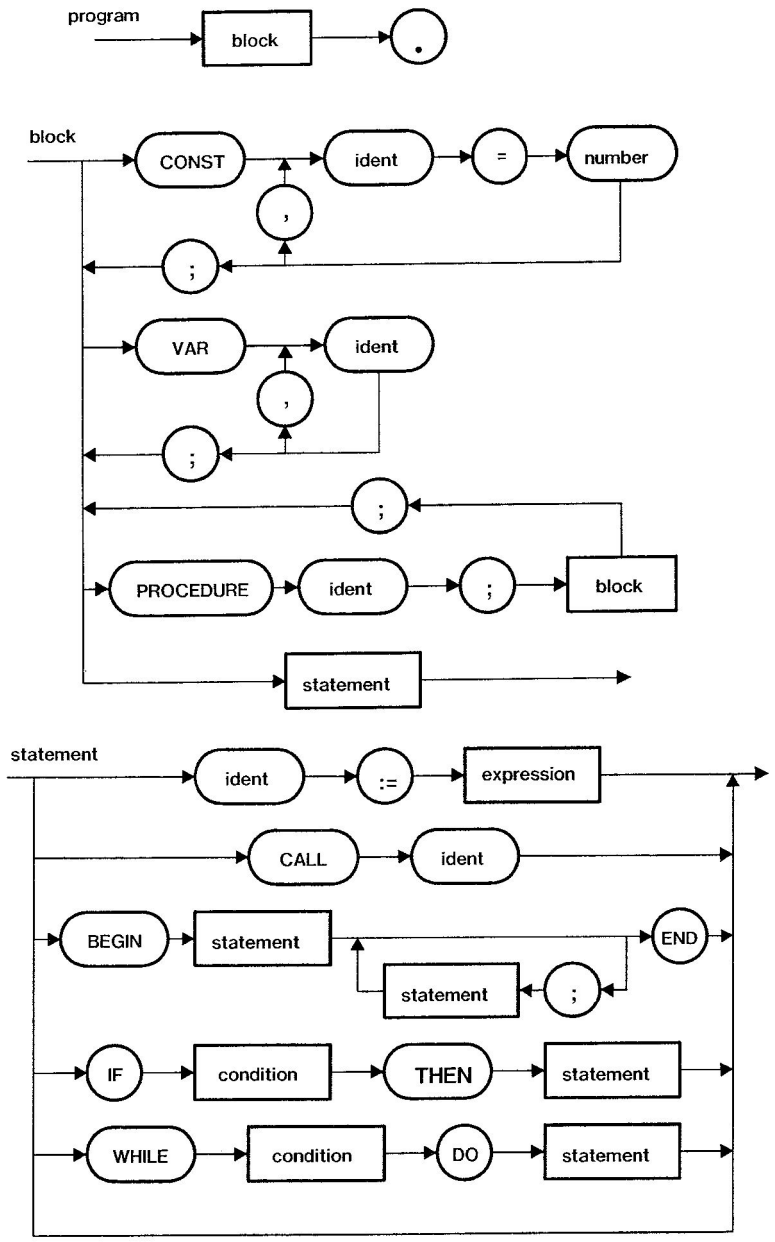
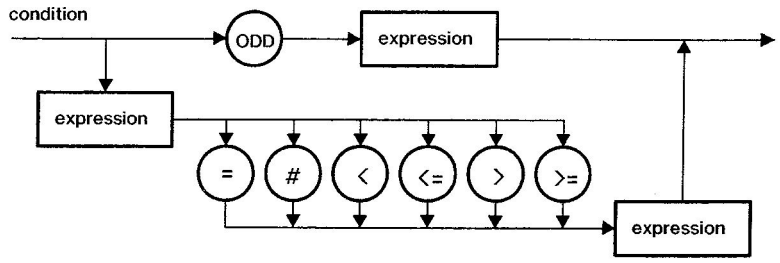
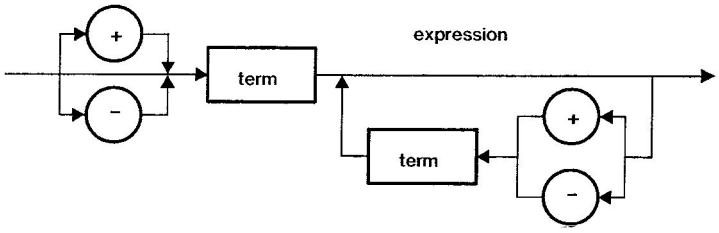
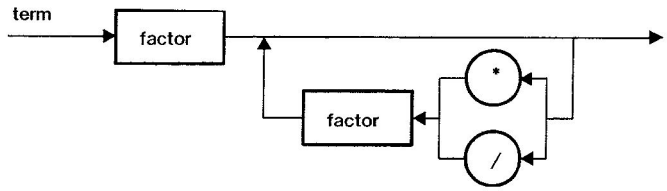
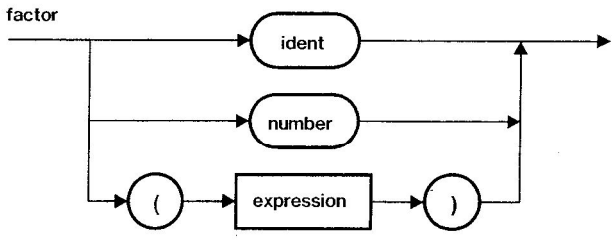


Fig.4. Syntaxdiagramme der Sprache PL/0


condition
ODD
expression
expression
=
#
<
<=
>
>=
expression


+
-
term
expression
term
+
-


term
factor
factor
*
/


factor
ident
number
(
expression
)

8. Ein Parser für PL/0.

In einem ersten Schritt soll ein Parser für PL/0 entwickelt werden, der alsdann nach der Methode des schrittweisen Ausbaus zu einem Compiler vervollständigt werden kann. Dieser erste Schritt kann strikte nach den angegebenen Regeln B1 bis B6 ausgeführt werden. Zuvor ist es aber nötig zu prüfen, ob die vorliegende Syntax den Regeln 1 und 2 aus Kapitel 2 genüge. Dies ist anhand der vorliegenden Graphen sehr leicht durchzuführen, sobald für jedes Diagramm (d.h. für jedes nicht-terminale Symbol) die Mengen der Anfangs- und der Folgesymbole ermittelt sind. Das Resultat dieser Ermittlung ist in folgender Tabelle festgehalten.

Symbol S	Anfangssymbole(S)	Folgesymbole(S)
Block	CONST VAR PROCEDURE ident ? ! CALL IF BEGIN WHILE	. ;
Statement	ident ? ! CALL BEGIN IF WHILE	. ; END
Condition	+ - (ident number	THEN DO
Expression	+ - (ident number	. ;) END THEN DO
Term	(ident number	. ;) END THEN DO
Factor	(ident number	. ;) END THEN DO

Anfangs- und Folgesymbole in PL/0

Der aufmerksame Leser wird bereits festgestellt haben, dass die Grundsymbole der Sprache PL/0 im Unterschied zu früheren Beispielen nicht nur aus einzelnen Zeichen bestehen, sodern im allgemeinen Zeichen*folgen* sind. So werden z.B. die Zeichenfolgen "BEGIN" und ":=", und sogar ganze Bezeichner als Symbole aufgefasst. Nun wird wie schon im Programm 3 ein Scanner eingesetzt, um einzelne Symbole aus der einzulesenden Zeichenfolge herauszulesen. Dieser Scanner verkörpert die sogenannten lexikographischen Regeln, welche bestimmen, wie Symbole aus einzelnen Zeichen aufgebaut sind. Wir nennen den Scanner wiederum *GetSym*. Er dient folgenden Zwecken:

1. Leerzeichen und Zeilenenden werden übersprungen.
2. Sogenannte *reservierte Wörter* wie BEGIN, END, etc. werden erkannt und als Symbole an den Parser weitergeleitet.

3. Folgen von Buchstaben und Ziffern, die nicht reservierte Wörter sind, werden als Bezeichner (identifiers) erkannt. Die Variable *sym* erhält den Wert *ident*. Der Bezeichner selbst wird der Variablen *id* zugewiesen.

4. Folgen von Ziffern werden als Zahlen erkannt. Die Variable *sym* erhält den Wert *number*, während die Zahl der Variablen *num* zugewiesen wird.

5. Kombinationen von Spezialzeichen (z.B. :=) werden erkannt und als Symbol weitergegeben.

6. Kommentare, dargestellt durch Zeichenfolgen, die mit (* beginnen und mit *) enden, werden übersprungen.

Der Scanner ist durch Programm 4 dargestellt. Er ist als Modul formuliert und ist ein klassisches Beispiel für die Anwendung des Modulkonzeptes. Es erlaubt, gewisse Einzelheiten vor den Benützern versteckt zu halten, und nur diejenigen Aspekte zur Schau zu stellen (zu *exportieren*), die für die Verwendung des Moduls relevant sind. Diese sind im sogenannten *Definitionsteil* zusammengefasst, der üblicherweise als eigenes Textfile gespeichert ist und sich separat compilieren lässt.

Die Prozedur *Getch* liefert das jeweils nächste Zeichen des Quelltextes und kopiert dieses in die Ausgabe. Zu diesem Zweck wird in unserem Beispiel nicht wie üblich eine weitere Datei herangezogen, sondern die Ausgabe erfolgt in ein sogenanntes *Fenster* (window). Wir gehen dabei von der Annahme aus, dass ein Bildschirm als Ausgabemedium zur Verfügung stehe, der von einem Modul unterstützt wird, welcher mehrere Ausgabeströme in verschiedene Fenster leitet. Obwohl hier nur ein einziger Ausgabestrom vorliegt, werden im Verlauf der weiteren Entwicklung des Compilers noch zusätzliche eingeführt werden. In Programm 4 sind die notwendigen Hilfsprozeduren aus einem hier nicht näher zu beschreibenden Modul *TextWindows* importiert. Die Lage und die Grösse eines Fensters werden bei dessen Eröffnung durch die Prozedur *OpenTextWindow* spezifiziert.

Die Prozedur *Mark(n)* dient zur Sichtbarmachung von entdeckten Fehlern. Der Einfachheit halber werden lediglich Fehlernummern in den widergegebenen Quelltext eingestreut, wobei noch die durch den Fenstermechanismus offerierte Möglichkeit ausgenutzt wird, die Nummern durch Inversion der Darstellung hervorzuheben (z.B. weiss auf schwarz anstatt schwarz auf weiss). Die Bedeutung der einzelnen Nummern ist aus der Tabelle am Ende von Kapitel 8 ersichtlich.

Der Scanner erfüllt die Aufgabe, genau ein Symbol im Text vorauszublicken. Die Entscheidungen des Parsers beruhen auf diesem nächsten Symbol, dargestellt durch die Variable *sym*. Zusätzlich blickt nun aber die Prozedur *GetCh* noch um ein weiteres Zeichen voraus. Dies ist hier notwendig, weil Symbole aus mehreren Zeichen bestehen können. Die Einzelheiten dieser Routinen sind aus dem Programm 4 ersichtlich, welches einen vollständigen Scanner für PL/0 darstellt.

```
DEFINITION MODULE PL0Scanner;
  FROM FileSystem IMPORT File;

  TYPE Symbol =
      (null, odd, times, div, plus, minus,
       eql, neq, lss, leq, gtr, geq,
       comma, rparen, then, do, lparen,
       becomes, number, ident, semicolon, end,
       call, if, while, begin, read, write,
       const, var, procedure, period, eof);

  VAR sym: Symbol;        (*last symbol read*)
    id: CARDINAL;         (*character buffer index*)
    num: CARDINAL;        (*last number read*)
    source: File;

  PROCEDURE Diff(u,v: CARDINAL): INTEGER;
    (*difference between identifiers at buf[u] and buf[v]*)
  PROCEDURE KeepId;
  PROCEDURE Mark(n: CARDINAL);  (*mark error n in source text*)
  PROCEDURE GetSym;  (*get next symbol; results: sym, id, num*)
  PROCEDURE InitScanner;
  PROCEDURE CloseScanner;
END PL0Scanner.

IMPLEMENTATION MODULE PL0Scanner;
  FROM Terminal IMPORT Read, BusyRead;
  FROM FileSystem IMPORT ReadChar;
  FROM TextWindows IMPORT Window, OpenTextWindow,
       Write, WriteCard, Invert, CloseTextWindow;

  CONST maxCard = 177777B; bufLen = 1000;
```

```
VAR ch: CHAR;          (*last character read*)
  id0, id1: CARDINAL;  (*indices to identifier buffer*)
  win: Window;
  keyTab: ARRAY [1..20] OF
          RECORD sym: Symbol; ind: CARDINAL END ;
  K: CARDINAL;         (*no of key words*)
  buf: ARRAY [0..bufLen-1] OF CHAR;
  (*character buffer;
    identifiers are stored with leading length count*)

PROCEDURE Mark(n: CARDINAL);
BEGIN Invert(win, TRUE);
  WriteCard(win, n, 1); Invert(win, FALSE)
END Mark;

PROCEDURE GetCh;
BEGIN ReadChar(source, ch); Write(win, ch)
END GetCh;

PROCEDURE Diff(u,v: CARDINAL): INTEGER;
  VAR w: CARDINAL;
BEGIN w := ORD(buf[u]);
  LOOP
    IF w = 0 THEN RETURN 0
    ELSIF buf[u] # buf[v] THEN
      RETURN INTEGER(buf[u]) - INTEGER(buf[v])
    ELSE u := u+1; v := v+1; w := w-1
    END
  END
END Diff;

PROCEDURE KeepId;
BEGIN id := id1
END KeepId;

PROCEDURE Identifier;
  VAR k, l, m: CARDINAL; d: INTEGER;
BEGIN id1 := id;
  IF id1 < bufLen THEN INC(id1) END;
  REPEAT
    IF id1 < bufLen THEN
```

```
      buf[id1] := ch; id1 := id1 + 1
    END;
    GetCh
  UNTIL (ch < "0") OR ("9" < ch) & (CAP(ch) < "A") OR
        ("Z" < CAP(ch));
  buf[id] := CHR(id1-id); (*Length*)
  k := 1; l := K;
  REPEAT m := (k + l) DIV 2;
    d := Diff(id, keyTab[m].ind);
    IF d <= 0 THEN l := m - 1 END;
    IF d >= 0 THEN k := m + 1 END
  UNTIL k > l;
  IF k > l + 1 THEN
    sym := keyTab[m].sym
  ELSE sym := ident
  END
END Identifier;

PROCEDURE Number;
  VAR i, j, k, d: CARDINAL;
  dig: ARRAY [0..31] OF CHAR;
BEGIN sym := number; i := 0;
  REPEAT dig[i] := ch; i := i+1; GetCh
  UNTIL (ch < "0") OR ("9" < ch) & (CAP(ch) < "A") OR
        ("Z" < CAP(ch));
  j := 0; k := 0;
  REPEAT d := CARDINAL(dig[j]) - 60B;
    IF (d < 10) & ((maxCard-d) DIV 10 >= k) THEN k := 10*k + d
      ELSE Mark(30); k := 0
    END ;
    j := j+1
  UNTIL j = i;
  num := k
END Number;

PROCEDURE GetSym;
  VAR xch: CHAR;

  PROCEDURE Comment;
  BEGIN GetCh;
    REPEAT
```

```
      WHILE ch # "*" DO GetCh END ;
      GetCh
    UNTIL ch = ")";
    GetCh
  END Comment;

BEGIN BusyRead(xch);
  IF xch > 0C THEN Read(xch) END ;
  LOOP (*ignore control characters*)
    IF ch <= " " THEN
      IF ch = 0C THEN ch := " "; EXIT END ;
      GetCh
    ELSIF ch >= 177C THEN GetCh
    ELSE EXIT
    END
  END ;
  CASE ch OF   (* " " <= ch < 177C *)
      " "  : sym := eof; ch := 0C |
      "!"  : sym := write; GetCh |
      '"'  : sym := null; GetCh |
      "#"  : sym := neq;  GetCh |
      "$"  : sym := null; GetCh |
      "%"  : sym := null; GetCh |
      "&"  : sym := null; GetCh |
      "'"  : sym := null; GetCh |
      "("  : GetCh;
             IF ch = "*" THEN Comment; GetSym
               ELSE sym := lparen
             END |
      ")"  : sym := rparen; GetCh|
      "*"  : sym := times; GetCh |
      "+"  : sym := plus; GetCh  |
      ","  : sym := comma; GetCh |
      "-"  : sym := minus; GetCh |
      "."  : sym := period; GetCh |
      "/"  : sym := div; GetCh |
      "0".."9": Number |
      ":"  : GetCh;
             IF ch = "=" THEN GetCh; sym := becomes
               ELSE sym := null
             END |
```

```
      ";"  : sym := semicolon; GetCh |
      "<"  : GetCh;
             IF ch = "=" THEN GetCh; sym := leq
               ELSE sym := lss
             END |
      "="  : sym := eql; GetCh |
      ">"  : GetCh;
             IF ch = "=" THEN GetCh; sym := geq
               ELSE sym := gtr
             END |
      "?"  : sym := read; GetCh  |
      "@"  : sym := null; GetCh  |
      "A".."Z": Identifier |
      "[".."`": sym := null; GetCh |
      "a".."z": Identifier |
      "{".."~": sym := null; GetCh
  END
END GetSym;

PROCEDURE InitScanner;
BEGIN ch := " ";
  IF id0 = 0 THEN id0 := id
    ELSE id := id0; Write(win, 14C)
  END
END InitScanner;

PROCEDURE CloseScanner;
BEGIN CloseTextWindow(win)
END CloseScanner;

PROCEDURE EnterKW(sym: Symbol; name: ARRAY OF CHAR);
VAR l, L: CARDINAL;
BEGIN K := K+1;
  keyTab[K].sym := sym;
  keyTab[K].ind := id;
  l := 0; L := HIGH(name);
  buf[id] := CHR(L+2);
  INC(id);
  WHILE l <= L DO
    buf[id] := name[l];
    id := id+1; l := l+1
```

```
    END
  END EnterKW;

BEGIN K := 0; id := 0; id0 := 0;
  EnterKW(do, "DO");
  EnterKW(if, "IF");
  EnterKW(end, "END");
  EnterKW(odd, "ODD");
  EnterKW(var, "VAR");
  EnterKW(call, "CALL");
  EnterKW(then, "THEN");
  EnterKW(begin, "BEGIN");
  EnterKW(const, "CONST");
  EnterKW(while, "WHILE");
  EnterKW(procedure, "PROCEDURE");
  OpenTextWindow(win, 0, 574, 704, 354, "PROGRAM");
END PL0Scanner.
```

Programm 4: Scanner für PL/0

Der Parser ist im Programm 5 wiedergegeben. Er ist sogar bereits erweitert worden und trägt gelesene Bezeichner in Tabellen ein, wenn sie in Vereinbarungen auftreten. Ihr Vorkommen in Anweisungen bewirkt sodann ein Suchen des Bezeichners in der Tabelle. Damit wird festgestellt, ob er ordnungsgemäss im Programm vereinbart ist. Das Fehlen einer Vereinbarung kann tatsächlich als eine Art syntaktischer Fehler betrachtet werden, da ein nicht vereinbarter Bezeichner als nicht zum Vokabular der Sprache gehörendes Symbol gilt. Der Umstand, dass dieser formale Fehler nur entdeckt werden kann, indem Information in Tabellen festgehalten wird, ist eine Manifestation der Tatsache, dass übliche Programmier-Sprachen nur in beschränktem Sinn kontext-frei sind (siehe Kapitel 1). Bezeichner enthalten eine Bedeutung, die nicht aus der Struktur des Textes allein, sondern vor allem aus ihrer Vereinbarung hervorgeht. Dennoch ist die kontext-freie Syntax ein äusserst nützliches Modell für diese Art von Sprachen. Sie ist durchaus geeignet, um einen einfachen, systematischen Aufbau von Übersetzern zu ermöglichen. Die so erhaltenen Grundstrukturen der Compiler können dann leicht erweitert werden, um die beschränkte Art der Kontext-Abhängigkeit zu verkraften. Die Tabelle der Bezeichner ist das einzige Instrument, das dazu nötig ist. Ihre Funktion ist leicht verständlich und kann durch wenige, isolierte Prozeduren ausgedrückt werden. Man beachte, dass im vorliegenden Fall die Tabelle als eine pulsierende Liste organisiert ist. Damit

wird in einfacher Weise den Regeln der Lokalität entsprochen. Beim Durchsuchen werden lokale Bezeichner zuerst gefunden. Am Ende einer Prozedurvereinbarung, welche einen Gültigkeitsbereich für Bezeichner darstellt, werden die lokalen Bezeichner leicht entfernt, da sie in der Liste zuoberst liegen.

Bevor die einzelnen Prozeduren des Parsers aufgebaut werden, ist es nützlich festzustellen, in welcher Weise sie aufeinander Bezug nehmen. Wir konstruieren dazu ein sogenanntes Bezugsdiagramm. Jeder Graph G wird darin als Knoten dargestellt, von dem ein Verweis (Zeiger) zu jedem Graphen G1 ... Gn ausgeht, auf die in der Definition von G Bezug genommen wird. Dementsprechend zeigt das Diagramm, welche Prozeduren G1 ... Gn von einer gegebenen Prozedur G aus aufgerufen werden.

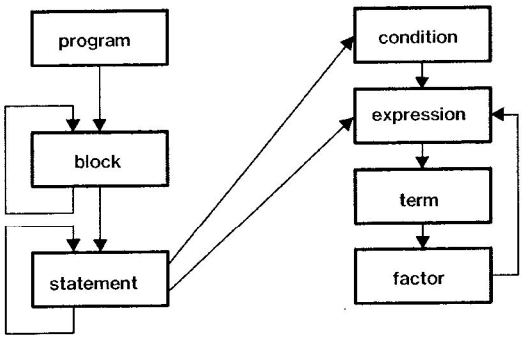


Fig. 5. Bezugsdiagramm der Parser-Prozeduren für PL/0

Jede Schleife im Bezugsdiagramm entspricht einer Rekursion. Es ist daher unerlässlich, dass für die Programmierung eine Sprache zur Verfügung steht, die rekursive Aufrufe von Prozeduren zulässt. Im weiteren zeigt das Bezugsdiagramm, auf welche Art die einzelnen Prozeduren lokal (verschachtelt) vereinbart werden können. Als einzige Struktur, auf die nie Bezug genommen wird, erscheint das Programm. Die entsprechende Prozedur kann daher ebensogut als Hauptprogramm des Compilers gelten.

```
DEFINITION MODULE PL0Parser;
  VAR noerr: BOOLEAN;
  PROCEDURE Parse;
  PROCEDURE EndParser;
END PL0Parser.
```

```
IMPLEMENTATION MODULE PL0Parser;
  FROM TextWindows IMPORT Window, OpenTextWindow, Write,
    WriteLn, WriteCard, WriteString, Invert, CloseTextWindow;
  FROM PL0Scanner IMPORT
       Symbol, sym, id, num, Diff, KeepId, GetSym, Mark;

  TYPE ObjectClass = (Const, Var, Proc, Header);
       ObjPtr = POINTER TO Object;
       Object = RECORD name: CARDINAL;
                  next:  ObjPtr;
                  CASE kind: ObjectClass OF
                    Const, Var, Proc: |
                    Header: last, down: ObjPtr
                  END
                END ;

  VAR topScope, bottom: ObjPtr;
      win: Window;

  PROCEDURE err(n: CARDINAL);
  BEGIN noerr := FALSE; Mark(n); Invert(win, TRUE);
    WriteCard(win, n, 1); Invert(win, FALSE)
  END err;

  PROCEDURE NewObj(k: ObjectClass): ObjPtr;
    VAR obj: ObjPtr;
  BEGIN (*enter new object into list*)
    ALLOCATE(obj, TSIZE(Object));
    WITH obj↑ DO
      name := id; kind := k; next := NIL
    END ;
    KeepId; topScope↑.last↑.next := obj; topScope↑.last := obj;
    RETURN obj
  END NewObj;

  PROCEDURE find(id: CARDINAL): ObjPtr;
    VAR hd, obj: ObjPtr;
  BEGIN hd := topScope;
    WHILE hd # NIL DO
```

```
    obj := hd↑.next;
    WHILE obj # NIL DO
      IF Diff(id, obj↑.name) = 0 THEN RETURN obj
        ELSE obj := obj↑.next
      END
    END ;
    hd := hd↑.down
  END ;
  err(11); RETURN NIL
END find;

PROCEDURE expression;

  PROCEDURE factor;
    VAR obj: ObjPtr;
  BEGIN WriteString(win, "factor"); WriteLn(win);
    IF sym = ident THEN obj := find(id);
      WITH obj↑ DO
        CASE kind OF
            Const, Var: |
            Proc:  err(21)
        END
      END ;
      GetSym
    ELSIF sym = number THEN GetSym
    ELSIF sym = lparen THEN
      GetSym; expression;
      IF sym = rparen THEN GetSym ELSE err(7) END
    ELSE err(8)
    END
  END factor;

  PROCEDURE term;
  BEGIN WriteString(win, "term"); WriteLn(win);
    factor;
    WHILE (times <= sym) & (sym <= div) DO
      GetSym; factor
      END
    END
  END term;
```

```
BEGIN WriteString(win, "expression"); WriteLn(win);
  IF (plus <= sym) & (sym <= minus) THEN
    GetSym; term
  ELSE term
  END ;
  WHILE (plus <= sym) & (sym <= minus) DO
    GetSym; term
  END
END expression;

PROCEDURE condition;
BEGIN WriteString(win, "condition"); WriteLn(win);
  IF sym = odd THEN
    GetSym; expression
  ELSE
    expression;
    IF (eql <= sym) & (sym <= geq) THEN
      GetSym; expression
    ELSE err(20)
    END
  END
END condition;

PROCEDURE statement;
  VAR obj: ObjPtr;
BEGIN WriteString(win, "statement"); WriteLn(win);
  IF sym = ident THEN
    obj := find(id); GetSym;
    IF sym = becomes THEN GetSym
    ELSE err(13)
    END ;
    expression
  ELSIF sym = call THEN
    GetSym;
    IF sym = ident THEN
      obj := find(id); GetSym
    ELSE err(14)
    END
  ELSIF sym = begin THEN
    GetSym; statement;
    WHILE sym = semicolon DO
```

```
      GetSym; statement
    END ;
    IF sym = end THEN GetSym ELSE err(17) END
  ELSIF sym = if THEN
    GetSym; condition;
    IF sym = then THEN GetSym ELSE err(16) END ;
    statement
  ELSIF sym = while THEN
    GetSym; condition;
    IF sym = do THEN GetSym ELSE err(18) END ;
    statement
  ELSIF sym = read THEN
    GetSym;
    IF sym = ident THEN obj := find(id)
    ELSE err(14)
    END ;
    GetSym
  ELSIF sym = write THEN
    GetSym; expression
  END
END statement;

PROCEDURE block;
  VAR hd, obj: ObjPtr;

  PROCEDURE ConstDeclaration;
    VAR obj: ObjPtr;
  BEGIN WriteString(win, "ConstDeclaration"); WriteLn(win);
    IF sym = ident THEN
      GetSym;
      IF sym = eql THEN GetSym;
        IF sym = number THEN
          obj := NewObj(Const); GetSym
        ELSE err(2)
        END
      ELSE err(3)
      END
    ELSE err(4)
    END
  END ConstDeclaration;
```

```
  PROCEDURE VarDeclaration;
    VAR obj: ObjPtr;
  BEGIN WriteString(win, "VarDeclaration"); WriteLn(win);
    IF sym = ident THEN
      obj := NewObj(Var); GetSym
    ELSE err(4)
    END
  END VarDeclaration;


BEGIN WriteString(win, "block"); WriteLn(win);
  ALLOCATE(hd, TSIZE(Object));
  WITH hd↑ DO
    kind := Header; next := NIL; last := hd;
    name := 0; down := topScope
  END ;
  topScope := hd;
  IF sym = const THEN
    GetSym; ConstDeclaration;
    WHILE sym = comma DO
      GetSym; ConstDeclaration
    END ;
    IF sym = semicolon THEN GetSym ELSE err(5) END
  END ;
  IF sym = var THEN
    GetSym; VarDeclaration;
    WHILE sym = comma DO
      GetSym; VarDeclaration
    END ;
    IF sym = semicolon THEN GetSym ELSE err(5) END
  END ;
  WHILE sym = procedure DO GetSym;
    IF sym = ident THEN GetSym ELSE err(4) END ;
    obj := NewObj(Proc);
    IF sym = semicolon THEN GetSym ELSE err(5) END ;
    block;
    IF sym = semicolon THEN GetSym ELSE err(5) END
  END ;
  statement;
  topScope := topScope↑.down
END block;
```

```
  PROCEDURE Parse;
  BEGIN noerr := TRUE; topScope := NIL;
    Write(win, 14C); ResetHeap(bottom);
    GetSym; block;
    IF sym # period THEN err(9) END
  END Parse;

  PROCEDURE EndParser;
  BEGIN CloseTextWindow(win)
  END EndParser;
BEGIN OpenTextWindow(win, 0, 66, 234, 508, "PARSE");
END PLOParser.
```

Programm 5: PL/0 Parser

1 Verwende "=" anstatt ":=" !
2 Nach "=" muss eine Zahl folgen.
3 Nach dem Bezeichner muss "=" folgen.
4 Nach CONST, VAR, oder PROCEDURE muss ein Bezeichner folgen.
5 Strichpunkt (oder Komma) fehlt.
6 Ein Ausdruck kann nicht mit diesem Symbol beginnen.
7 Schliessende Klammer fehlt.
8 Der vorangehende Faktor kann nicht mit diesem Symbol enden.
9 Hier wird ein Punkt erwartet.
10 Inkorrektes Symbol in einer Anweisung.
11 Dieser Bezeichner ist nicht vereinbart.
12 Zuweisung zu Konstanten und Prozeduren sind nicht erlaubt.
13 Zuweisungsoperator ist ":=".
14 Nach "call" muss ein Bezeichner folgen.
15 Aufruf (call) einer Konstanten oder Variablen ist nicht erlaubt.
16 Hier wird THEN erwartet.
17 Hier wird ";" oder END erwartet.
18 Hier wird DO erwartet.
19 Auf diese Anweisung folgt ein inkorrekt verwendetes Symbol.
20 Hier wird eine Relation erwartet.
21 Ein Ausdruck darf keinen Prozedurbezeichner enthalten.
25 Ein Bezeichner darf nur einmal vereinbart werden.
30 Zahl ist zu gross.

Fehlermeldungen des PL/0 Compilers nach Programmen 5 und 8.

9. Die Behandlung von syntaktischen Fehlern.

Bisher erfüllte der Parser nur die eher bescheidene Aufgabe zu bestimmen, ob eine Folge von Symbolen einen syntaktisch korrekten Satz darstelle. Fast als Nebenprodukt entdeckt dabei der Parser die Struktur des gelesenen Textes. Sobald jedoch ein inkorrektes Symbol vorliegt, ist des Parsers Aufgabe erfüllt, und der Vorgang der Satzzerlegung kann abgebrochen werden. Für praktische Zwecke ist dieses Vorgehen natürlich unzulässig. Vielmehr muss ein Compiler in dieser Lage eine entsprechende Fehleranzeige ausgeben und danach den Zerlegungsprozess fortsetzen. Dabei ist es sogar wahrscheinlich, dass weitere Fehler entdeckt werden. Eine Fortsetzung des Prozesses ist aber nur möglich, wenn gewisse Annahmen oder Hypothesen über die Art des Fehlers gemacht werden. Je nach dieser Annahme muss ein gewisser Teil des Textes übersprungen oder eingefügt werden. Solche Annahmen sind notwendig, selbst wenn gar nie die Absicht gehegt wird, das fehlerhafte Programm zu korrigieren und dennoch auszuführen. Ohne eine einigermassen zutreffende Hypothese ist eine Fortsetzung der Satzzerlegung schlechthin unmöglich.

Die Technik der Wahl guter Hypothesen ist recht kompliziert. Sie beruht eher auf Heuristiken, da es bisher kaum gelungen ist, das Problem befriedigend zu formalisieren. Der Hauptgrund dabei ist, dass die rein formale Syntax viele Faktoren ausser Acht lässt, die für die Erkennung eines Satzes durch den Menschen wichtig sind. So ist z.B. das Fehlen einer Interpunktion ein sehr häufiger Fehler, und zwar nicht nur in Programmiersprachen. Hingegen ist es eher selten, dass in einem arithmetischen Ausdruck ein Pluszeichen verloren geht. Für den Parser sind jedoch sowohl Semicolon wie Pluszeichen Symbole ohne Klassenunterschied. Dem Programmierer aber erscheint das Semicolon als beinahe redundant, besonders am Zeilenende, während über die Wichtigkeit des Pluszeichens kein Zweifel besteht. Diese Art von Unterschieden muss aber in Betracht gezogen werden, wenn ein System sich in Bezug auf Fehler "vernünftig" verhalten soll. Daraus geht hervor, dass die Fehlerbehandlung entscheidend von der konkreten Sprache abhängt, und daher nur beschränkt durch allgemeingültige Regeln beschrieben werden kann.

Dennoch gibt es einige heuristische Regeln, die über den unmittelbaren Bereich von PL/0 hinaus Gültigkeit zu haben scheinen. Bezeichnenderweise betreffen sie den Entwurf der Sprache und deren Syntax ebensosehr wie die

Technik der Fehlerbehandlung selbst. Vor allem steht es ausser jedem Zweifel, dass *eine einfache Sprachstruktur eine vernünftige Fehlerbehandlung entscheindend erleichtert* oder sogar überhaupt erst ermöglicht.

Falls nach der Identifizierung eines Fehlers ein Teil des nachfolgenden Textes übersprungen werden soll, so ist es unerlässlich, dass die Sprache einige *Schlüsselwörter* enthält, deren Missbrauch sehr unwahrscheinlich ist. Sie sind daher zuverlässige Instrumente, um den Parser wieder korrekt zu orientieren. PL/0 macht sich diese Erkenntnis in hohem Masse zu Nutzen, indem jede Anweisung (ausser der Wertzuweisung) mit einem distinkten Schlüsselwort beginnt: BEGIN, WHILE, IF, etc. Dasselbe gilt für Vereinbarungen, die mit CONST, VAR, oder PROCEDURE beginnen.

Eine zweite Regel betrifft den Aufbau des Parsers in unmittelbarerer Weise. Beim top-down Parsing werden typischerweise Zerlegungsziele in Unterziele aufgeteilt, die vom Parser rekursiv in Angriff genommen werden. Diese zweite Regel besagt, dass im Fall eines Fehlers der Parser sein Ziel nicht einfach aufgeben und den Fehler an die ihn aufrufende Prozedur mitteilen soll, sondern dass der gegenwärtige Parser den Text selber soweit überspringen oder ergänzen soll, dass der aufrufende Parser die Zerlegung in plausibler Weise fortsetzen kann. Die programmiertechnische Konsequenz dieser Regel ist, dass jede Prozedur stets regulär zu Ende geführt wird, d.h. dass sie *keinen speziellen Fehlerausgang* besitzt.

Eine mögliche Interpretation dieser Regel führt dazu, dass in jedem Fall nach einem Fehler der Eingabetext so weit übersprungen wird, bis ein legales Folgesymbol vorliegt. Dies bedingt, dass jede Parserroutine die Menge der am Ort ihres Aufrufs gültigen Folgesymbole kennt. Diese Menge kann der Prozedur mittels eines zusätzlichen Parameters mitgeteilt werden Am Ende jeder Prozedur wird ein Test eingefügt, der feststellt, ob das nächste Symbol in dieser Menge enthalten sei. Der Test kann weggelassen werden, falls diese Bedingung bereits aus der Logik des bestehenden Parsers hervorgeht.

Es wäre jedoch sehr kurzsichtig, den Text unter allen Umständen bis zum nächsten legalen Folgesymbol zu überspringen. So ist es doch auch möglich, dass gerade dieses Folgesymbol irrtümlicherweise fehlt. Man denke an das häufige Beispiel eines fehlenden Strichpunktes! Das Überspringen des Textes kann in solchen Fällen unannehmbare Folgen haben. Wir ergänzen daher diese Mengen von Folgesymbolen durch zusätzliche Symbole, die zwar nicht legale Folgesymbole sind, aber auf jeden Fall das ziellose Überspringen abbremsen

sollen. Es ist unter diesen Umständen angezeigt, diese Symbole nicht mehr Folgesymbole, sondern *Stopsymbole* zu nennen. Es liegt auf der Hand, dass diese zusätzlichen Symbole vor allem die erwähnten Schlüsselsymbole sind, die am Anfang wichtiger Satzkonstruktionen stehen.

Aus Gründen der Flexibilität führen wir eine allgemein verwendbare Hilfsprozedur *test* ein, die in der erwähnten Art und Weise eingesetzt wird. Sie besitzt drei Parameter und ist nachfolgend beschrieben. Der Parameter s1 bestimmt die legalen Folgesymbole. Ist das nächste Symbol nicht in dieser Menge enthalten, so liegt ein syntaktischer Fehler vor. Der Parameter s2 gibt die zusätzlichen Stopsymbole an. Die Anwesenheit eines dieser Symbole ist unzweifelhaft als Fehler zu betrachten; dennoch darf dieses Symbol nicht übersprungen werden. Der dritte Parameter n gibt die Fehlernummer an, welche diesem Fall zugeordnet ist.

```
TYPE symset = SET OF Symbol;
PROCEDURE test(s1, s2: symset;  n: CARDINAL);
BEGIN
   IF NOT (sym IN s1) THEN
      error(n); s1 := s1 + s2;
      WHILE NOT (sym IN s1) DO GetSym END
   END
END test
```

Es zeigt sich, dass diese Prozedur sich auch vorteilhaft am Eingang eines Parsers verwenden lässt um festzustellen, ob das nächste Symbol ein legales Anfangssymbol der zu untersuchenden Struktur sei. Diese Vorkehrung ist an allen Stellen empfohlen, wo ein Parser aufgerufen wird ohne vorangehende Prüfung des ersten Symbols. In diesen Fällen wird für s1 die Menge der Anfangssymbole und für s2 die Menge der Folgesymbole eingesetzt.

Falls man ohne Verwendung der Mengenstruktur auskommen will (oder muss), so versucht man, die Symbolmenge derart zu ordnen, dass relevante Teilmengen durch Vergleichsoperatoren abgegrenzt werden können. Dieser Weg wird im vorliegenden PL/0 Compiler beschritten. Die Prozedur, deren Verwendung aus Programm 8 (siehe *factor* und *statement*) ersichtlich ist, lautet

```
PROCEDURE test(s: Symbol; n: CARDINAL);
BEGIN
   IF sym < s THEN error(n);
      REPEAT GetSym UNTIL sym >= s
```

```
    END
  END test
```

wobei s ein Grenzsymbol bezeichnet. Die Ordnung der Symbolmenge wird so gewählt, dass schwache Symbole, d.h. solche, die innerhalb von Ausdrücken vorkommen, am Anfang stehen, während starke Symbole, die grössere Einheiten wie Anweisungen und Blöcke trennen oder eröffnen, am Schluss folgen.

Das bis hierher entwickelte Schema beruht auf dem Prinizip, fehlerhafte Ausschnitte eines Textes zu überspringen und den Zerlegungsprozess an geeigneter Stelle wieder aufzunehmen. Dies ist aber eine unzweckmässige Strategie in all jenen Fällen, wo der Fehler darin besteht, dass ein einziges Symbol weggelassen wurde. Die Erfahrung zeigt, dass dies vor allem bei Symbolen mit rein syntaktischer Funktion vorkommt, bei Symbolen also, die nicht selbst irgend eine Operation spezifizieren. Eine hinreichende Lösung besteht darin, dass in diesen isolierten Fällen die Syntax der Sprache selbst erweitert wird. Dies geschieht so, dass das Fehlen des besagten Symbols gleichsam legalisiert wird, wobei natürlich der Parser beim Traversieren des neuen Pfades eine Fehlermeldung ausgibt. Als Beispiel für eine solche Erweiterung der Syntax sei die Struktur der Anweisungsfolge aufgeführt, die es nun gestattet, das Semicolon wegzulassen.

statement = ... | "BEGIN statement {[";"] statement} "END" | ...

Die Prozedur, die dieser Struktur zugeordnet ist, lässt sich in üblicher Weise durch die Regeln B1-B6 herleiten. Im nachfolgenden Programmstück wird allerdings eine Vereinfachung vorgenommen, indem eine loop-Anweisung verwendet wird. Dieselbe Konstellation findet sich in den Konstanten- und Variablen-Vereinbarungen, wo auf analoge Weise fehlende Kommata "eingefügt" werden.

```
IF sym = begin THEN GetSym;
  LOOP statement;
    IF sym = semicolon THEN GetSym
    ELSIF sym = end THEN GetSym; EXIT
    ELSIF sym < const THEN err(17)
    ELSE err(17); EXIT
  END
END
```

Es dürfte aus den vorangegangenen Ausführungen ersichtlich sein, dass es

keine perfekte Strategie der Fehlerbehandlung gibt, die alle korrekten Sätze mit hinreichender Effizienz übersetzt und alle inkorrekten Sätze vernünftig diagnostiziert. Jede Strategie wird gewisse abstruse Symbolfolgen in einer Weise behandeln, die ihrem Autor unerwartet erscheinen. Das wesentliche Merkmal eines guten Compilers jedoch ist, dass (1) keine Symbolfolge zum Zusammenbruch des Compilers führen kann, und (2) dass häufige, echte Fehler korrekt diagnostiziert werden und keine (oder wenige) zusätzliche, spätere Fehlermeldungen verursachen. Die hier vorgetragene Strategie arbeitet zufriedenstellend, obwohl es selbstverständlich Möglichkeiten zur Verbesserung gibt (s. Fig. 10). Bemerkenswert ist an dieser Methode, dass der verbesserte Compiler systematisch durch Anwendung einiger weniger Grundregeln aus dem reinen Parser entwickelt werden kann. Diese Regeln werden lediglich durch geschickte Wahl einiger Parameter ergänzt, die auf Erfahrung im Gebrauch der Sprache beruht. Die gemachten Erweiterungen werden beim Vergleichen der Programme 5 und 8 ersichtlich.

10. Ein Interpreter für PL/0.

Es ist in der Tat bemerkenswert, dass ein PL/0 Compiler bis hierher ohne Kenntnis des Computers entwickelt wurde, für den er Programme übersetzen soll. Aber aus welchem Grunde sollte die Struktur der Zielmaschine die Strategie der syntaktischen Analyse und der Fehlerbehandlung beeinflussen? Ganz im Gegenteil muss eine solche Beeinflussung sogar vermieden werden. Anstatt dessen wird eine geeignete Art der Code Generierung für einen beliebigen Computer nach dem Prinzip des schrittweisen Ausbaus auf das bestehende Gerüst des Compilers superponiert. Bevor wir diesen Schritt in Angriff nehmen können, ist es allerdings nötig, dass wir uns auf eine Zielmaschine und damit eine Zielsprache der Übersetzung festlegen.

Um den Compiler verhältnismässig einfach gestalten zu können und die Entwicklung nicht mit unwichtigen Einzelheiten zu belasten, die lediglich durch bestimmte Eigenschaften des gewählten Computers bedingt sind, postulieren wir einen Computer nach unserer eigenen Wahl. Daraus ergibt sich der beträchtliche Vorteil, dass er auf die Bedürfnisse der Quellsprache PL/0 zugeschnitten werden kann. Dieser Computer existiert zwar nicht als reale Maschine; wir nennen ihn daher hypothetisch. Da ein Computer jedoch stets Befehle nach einem bestimmten Rezept (Algorithmus) ausführt, lässt er sich durch ein Programm beschreiben. Ein realer Computer kann sodann benützt werden, um gemäss diesem Programm PL/0-Code zu interpretieren; daher wird es ein *Interpreter* genannt. Der hypothetische Computer wird also durch den Interpreter *emuliert*; er besitzt daher mindestens eine semi-reale Existenz.

Es ist nicht das Ziel dieser Ausführungen, auf die Beweggründe einzugehen, die zur Wahl der nachfolgend beschriebenen Computerstruktur und ihren Einzelheiten geführt haben. Vielmehr soll dieses Kapitel als eine Art beschreibendes Manual dienen, das aus einer informalen Einführung und einer formalen Definition des Computers durch das Programm des Interpreters besteht. Diese Formalisierung mag sogar als einfaches Beispiel für eine exakte, algorithmische Beschreibung von Prozessoren aufgefasst werden.

Der PL/0 Computer besteht aus zwei Speichern, einem Befehlsregister und drei Adressregistern. Der *Programmspeicher* (genannt *code*) wird durch den Compiler geladen und bleibt während der Ausführung eines Programmes

unverändert. Der *Datenspeicher S* wird durchwegs als Stapel (stack) verwendet. Alle arithmetischen Operatoren finden ihre Operanden zuoberst auf dem Stapel und ersetzen sie durch das Resultat. Das oberste Element wird durch das Index- bzw. Adressregister *T* (top) gekennzeichnet. Das *Befehlsregister I* enthält den gegenwärtig interpretierten Befehl. Der *Programmzähler P* enthält die Adresse bzw. den Index des nächsten Befehls, der zur Interpretation aus dem Programmspeicher zu holen ist.

Jede Prozedur in PL/0 kann lokale Variablen besitzen. Da Prozeduren rekursiv aufrufbar sind und jede Rekursion eigene lokale Variablen erheischt, ist es unmöglich, lokalen Variablen bereits bei der Compilation feste Speicherplätze zuzuweisen. Anstatt dessen wird während der Ausführung bei jedem Aufruf ein entsprechender Speicherplatz für den Satz lokaler Variablen festgelegt. Am Ende der Prozedur wird er wieder freigegeben. Weil jede Prozedur Q, die nach einer Prozedur P aufgerufen wurde, beendet wird, bevor P beendet ist (last in first out), ist die Speicherverwaltung nach dem Stapelprinzip die naheliegende Lösung. Beim Aufruf wird der Prozedur zuoberst im Stapel ein Platz zugewiesen; er wird *Prozedur-Segment* oder auch *activation record* genannt. Jede Prozedur speichert darin zuerst einige "private" Daten ab, zumindest die Programmadresse ihres Aufrufs (oft *return address* genannt) und die Segment-Adresse der aufrufenden Prozedur. Diese beiden Daten sind unerlässlich, um nach Beendigung der Prozedur den vor dem Aufruf geltenden Zustand wieder herstellen zu können. Wir fassen sie als implizite lokale Daten einer Prozedur auf. Sie stellen den sogenannten *Segment-Descriptor* dar mit den zwei Komponenten RA (*return address*) und DL (*dynamic link*). Die DL-Werte verketten alle Segmente im Stapel; der Anfang dieser Kette befindet sich im Basis-Adress Register B (siehe Fig. 6).

Da die eigentliche Speicherplatzzuweisung erst während der Interpretation des compilierten Programmes stattfindet, kann der Compiler keine endgültigen Adressen in die Befehle einbauen. Er kann lediglich den Platz einer Variablen innerhalb eines Segmentes bestimmen. Diese relative Adresse wird *offset* genannt. Der Interpreter muss sodann, um eine absolute Adresse zu erhalten, die Basis-Adresse des betreffenden Segmentes zum offset hinzuzählen. Wenn eine Variable lokal zur gegenwärtig ausgeführten Prozedur ist, so kann diese Basis-Adresse dem Register B entnommen werden. Ansonst muss sie ermittelt werden, indem die Adress-Kette DL bis zum betreffenden Segment durchlaufen wird. Der Compiler kennt jedoch nur die Tiefe der statischen Verschachtelung von Prozeduren, während die Kette DL die dynamische Folge von Aufrufen festhält. Leider sind die beiden Folgen nicht notwendigerweise identisch. Dies wird durch das folgende Beispiel veranschaulicht.

Wir nehmen an, dass eine Prozedur A eine Prozedur B aufruft, die lokal zu A ist. In gleicher Weise ruft B eine Prozedur C, die lokal zu B vereinbart ist. Schliesslich ruft C die Prozedur B (rekursiv). Wir sagen, dass A auf Stufe 1, B auf 2, und C auf Stufe 3 vereinbart sind. Findet nun in B ein Zugriff zu einer Variablen von A statt, so weiss der Compiler, dass ein Stufen-Unterschied von 1 besteht. Ein Zugriff entlang der *dynamischen Kette* DL hingegen führt im ersten Schritt in das Segment der Prozedur C. Aus diesem Beispiel geht hervor, dass eine zweite Verkettung nötig ist, welche den Zugriffspfad zu Variablen korrekt widerspiegelt (siehe Fig. 6). Wir nennen diese zweite Kette die *statische Kette* SL.

Adressen werden deshalb vom Compiler als Zahlenpaare dargestellt. Die erste Zahl bezeichnet die Stufen-Differenz zwischen dem Aufrufer und der zugegriffenen Variablen und bestimmt die Anzahl der in der SL-Kette zu durchlaufenden Glieder. Die zweite Zahl ist der offset, d.h. die Adresse relativ zum Anfang des die Variable enthaltenden Segmentes.

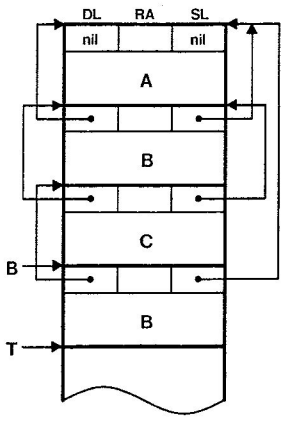


Fig. 6. Stapel und Segment-Verkettung

Der Befehlssatz des PL/0 Computers ist direkt auf die Erfordernisse der Sprache zugeschnitten. Er enthält die folgenden Befehle:

1. Ein Befehl LIT, um Zahlen (literals) in den Stapel zu bringen.
2. Ein Befehl LOD, um Variablenwerte in den Stapel zu laden.
3. Ein Befehl STO, um Werte aus dem Stapel abzuspeichern (store).
4. Ein Befehl CAL, um eine Prozedur aufzurufen (call).

5. Ein Befehl INT, um Speicher zu reservieren (increment T).
6. Befehle JMP und JPC, um im Programm zu springen (jump).
7. Ein Satz von arithmetischen und relationalen Operatoren OPR.

Das Befehlsformat wird durch die Notwendigkeit dreier Befehlskomponenten bestimmt. Jeder Befehl enthält einen Befehlscode f, der die Art des Befehls kennzeichnet, sowie einen (oder zwei) Parameter. Im Fall der Operatoren gibt der Parameter die Identität des Operators an, da der Befehlscode lediglich eine Befehlsklasse bezeichnet. Bei den Befehlen LIT und INT ist der Parameter eine Zahl, bei JMP, JPC und CAL eine Programmadresse, bei LOD und STO eine Datenadresse.

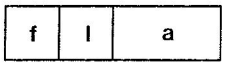


Fig. 7. Befehlsformat des PL/0 Computers

Die Einzelheiten und die genaue Funktionsweise des PL/0 Computers sind dem Programm 6 zu entnehmen. Es enthält den Interpreter als Prozedur, welche aufgerufen wird, nachdem der Compiler den zu interpretierenden Code erzeugt hat. Das Programm stellt sich wie der Scanner als separat formulierter Modul dar, der die Prozedur *Interpret* exportiert. Die Ausgabe von Daten fliesst in ein Fenster, das dem Interpreter zueigen ist. Die Prozedur *EndInterpreter* dient dazu, das Fenster zu schliessen.

```
DEFINITION MODULE PLOInterpreter;
  CONST maxfct = 15;
        maxlev = 15;
        maxadr = 1023;

  TYPE Instruction = RECORD f: [0..maxfct];  (*function*)
                            l: [0..maxlev];  (*level*)
                            a: [0..maxadr]   (*address*)
                     END ;

  VAR code: ARRAY [0..maxadr] OF Instruction;

  PROCEDURE Interpret;
  PROCEDURE EndInterpreter;
END PLOInterpreter.
```

```
IMPLEMENTATION MODULE PLOInterpreter;
  FROM TextWindows IMPORT Window, Done, OpenTextWindow,
    ReadInt, Write, WriteLn, WriteInt, Invert, CloseTextWindow;

  VAR  win: Window;

  PROCEDURE Interpret;
    CONST stacksize = 1000;
    VAR p,b,t: INTEGER;   (*Program-, Base-, Stack-Registers*)
        i: Instruction;   (*instruction register*)
        s: ARRAY [0..stacksize-1] OF INTEGER;   (*data store*)

    PROCEDURE base(l: INTEGER): INTEGER;
      VAR b1: INTEGER;
    BEGIN b1 := b;    (*find base address, l levels down*)
      WHILE l > 0 DO
        b1 := s[b1]; l := l-1
      END ;
      RETURN b1
    END base;

  BEGIN (*interpret*) Write(win, 14C);
    t := 0; b := 1; p := 0;
    s[1] := 0; s[2] := 0; s[3] := 0;
    REPEAT i := code[p]; p := p+1;
      WITH i DO
        CASE f OF
         0: t := t+1; s[t] := a |
         1: CASE a OF  (*operators*)
              0: (*RET*) t := b-1; p := s[t+3]; b := s[t+2] |
              1: s[t] := -s[t] |
              2: t := t-1; s[t] := s[t] + s[t+1] |
              3: t := t-1; s[t] := s[t] - s[t+1] |
              4: t := t-1; s[t] := s[t] * s[t+1] |
              5: t := t-1; s[t] := s[t] DIV s[t+1] |
              6: s[t] := ORD(ODD(s[t])) |
              7: |
              8: t := t-1; s[t] := ORD(s[t] = s[t+1])  |
              9: t := t-1; s[t] := ORD(s[t] # s[t+1])  |
```

```
            10: t := t-1; s[t] := ORD(s[t] < s[t+1])  |
            11: t := t-1; s[t] := ORD(s[t] >= s[t+1]) |
            12: t := t-1; s[t] := ORD(s[t] > s[t+1])  |
            13: t := t-1; s[t] := ORD(s[t] <= s[t+1]) |
            14: t := t+1; Write(win, ">");
                Invert(win, TRUE); ReadInt(win, s[t]);
                IF NOT Done THEN p := 0 END ;
                Invert(win, FALSE) |
            15: WriteInt(win, s[t], 7); WriteLn(win); t := t-1
            END |
         2: t := t+1; s[t] := s[base(l)+INTEGER(a)] |
         3: s[base(l)+INTEGER(a)] := s[t]; t := t-1 |
         4: (*generate new block mark*)
            s[t+1] := base(l); s[t+2] := b; s[t+3] := p;
            b := t+1; p := a |
         5: t := t + INTEGER(a) |
         6: p := a |
         7: IF s[t] = 0 THEN p:= a END ;
            t := t-1
       END
     END
   UNTIL p = 0
 END Interpret;

 PROCEDURE EndInterpreter;
 BEGIN CloseTextWindow(win)
 END EndInterpreter;

BEGIN OpenTextWindow(win, 470, 66, 234, 508, "RESULT")
END PLOInterpreter.
```

Programm 6: Interpreter für PL/0-Code

11. Die Erzeugung von Befehls-Code.

Analog zur Eingabe des Quelltextes, wo die Umwandlung von Zeichen zu Symbolen von einem Scanner übernommen wird, kann bei der Ausgabe des Zielcodes die administrative Aufgabe der Abspeicherung von einem Generator übernommen werden. Dieser wird zweckmässigerweise wiederum als separater Modul formuliert (siehe Programm 7) und bezieht sich auf das durch den Interpreter definierte Befehlsformat. Er stellt eine Prozedur Gen(f,l,a) zur Verfügung und reiht den durch die Parameter bestimmten Befehl der bereits gespeicherten Befehlsfolge an.

```
DEFINITION MODULE PL0Generator;
  PROCEDURE Label(): CARDINAL;      (*label of next instruction*)
  PROCEDURE Gen(f,l,a: CARDINAL);   (*generate instruction*)
  PROCEDURE fixup(x: CARDINAL);     (*fixup code[x]*)
  PROCEDURE InitGenerator;
  PROCEDURE EndGenerator;
END PL0Generator.

IMPLEMENTATION MODULE PL0Generator;
  FROM TextWindows IMPORT Window, OpenTextWindow,
       WriteString, Write, WriteLn, WriteCard, CloseTextWindow;
  FROM PL0Interpreter IMPORT Instruction, maxfct, maxadr, code;

  VAR L: CARDINAL;  (*current label*)
      win: Window;
      mnemonic: ARRAY [0..maxfct], [0..3] OF CHAR;

  PROCEDURE InitGenerator;
  BEGIN L := 0; Write(win, 14C)
  END InitGenerator;

  PROCEDURE Label(): CARDINAL;
  BEGIN RETURN L
  END Label;
```

```
  PROCEDURE Gen(x, y, z: CARDINAL);
  BEGIN
    IF L >= maxadr THEN HALT END ;
    WITH code[L] DO
      f := x; l := y; a := z
    END ;
    WriteCard(win, L, 4);
    WriteString(win, mnemonic[x]);
    WriteCard(win, y, 3); WriteCard(win, z, 6); WriteLn(win);
    L := L+1
  END Gen;

  PROCEDURE fixup(x: CARDINAL);
  BEGIN code[x].a := L; WriteString(win, "fixup at");
    WriteCard(win, x, 4); WriteLn(win)
  END fixup;

  PROCEDURE EndGenerator;
  BEGIN CloseTextWindow(win)
  END EndGenerator;

BEGIN OpenTextWindow(win, 235, 66, 234, 508, "CODE");
  mnemonic[0] := " LIT"; mnemonic[1] := " OPR";
  mnemonic[2] := " LOD"; mnemonic[3] := " STO";
  mnemonic[4] := " CAL"; mnemonic[5] := " INT";
  mnemonic[6] := " JMP"; mnemonic[7] := " JPC";
END PL0Generator.
```

Programm 7: Code-Generierungs Modul für PL/0

Um Befehle erzeugen zu können, muss der Compiler sowohl die Art des Befehls wie auch dessen Parameter kennen. Die Werte der Parameter, die sich auf Variablen, Konstanten und Prozeduren beziehen, sind mit Bezeichnern verknüpft. Die Parameter werden während der Übersetzung von Vereinbarungen ermittelt und in der Datenstruktur des Compilers eingetragen. Die Struktur der Tabelle wird erweitert (siehe Programm 5), um die relevanten Attribute der Bezeichner festzuhalten. Identifiziert der Bezeichner eine Konstante, so ist sein Attribut eine Zahl; bei einer Variablen ist es eine Datenadresse (bestehend aus einem Zahlenpaar), bei einer Prozedur sind es eine Programmadresse und die Schachtelungstiefe der Vereinbarung. Die entsprechende Erweiterung der Struktur der Datenstruktur des Parsers ist dem Programm 8 zu entnehmen. Sie ist

ein bemerkenswertes Beispiel eines schrittweisen Ausbaus nicht nur des Programms, sondern auch der zugehörigen Datenstruktur.

Während bei der Verarbeitung von Konstanten-Vereinbarungen der Wert der Konstanten direkt dem Programm entnommen werden kann, müssen bei den Variablen-Vereinbarungen die Adressen vom Compiler berechnet werden. Dieser Prozess ist im allgemeinen nicht einfach. PL/0 jedoch ist genügend primitiv, um sequentielle Speicherzuweisung zu ermöglichen. Dies besagt, dass ein Adresszähler, der den Platz einer Variablen bestimmt, bei jeder Vereinbarung einfach um 1 erhöht wird. Dies ist daher zugelassen, weil jede Variable in der PL/0 Maschine genau eine Speicherzelle beansprucht. Der Zähler *adr* muss lokal zur Prozedure block vereinbart sein, da er Adressen relativ zum jeweiligen Segmentanfang angibt (siehe Kapitel 10). Bei jeder Vereinbarung wird die Prozedur *enter* aufgerufen, welche die notwendigen Eintragungen in der Datenstruktur vornimmt.

Unter diesen Umständen ist es verhältnismässig einfach, Programme in Befehlsfolgen zu übersetzen. Dank der Organisation des Datenspeichers als Stapel gilt dies ganz besonders für Ausdrücke. Die Hauptaufgabe des Compilers ist das Umordnen von Operanden und Operatoren in die sogenannte *Postfix*-Notation. Darin *folgt* ein Operator stets seinen Operanden, anstatt wie bei der *Infix* Notation zwischen ihnen eingebettet zu sein. Die Postfix-Form wird oft auch *klammerfrei* genannt, weil sie gestattet, Ausdrücke ohne Klammern eindeutig darzustellen. Einige Infix-Ausdrücke und ihre entsprechenden Postfix- Formen stellen diese Eigenschaft unter Beweis:

x + y	x y +
(x-y) + z	x y - z +
x - (y+z)	x y z + -
(x+y)/(z-w)	x y + z w - /

Die Übersetzungsregeln für die syntaktischen Einheiten des Ausdrucks, Terms und Faktors lassen sich wie folgt schematisieren, wobei T(S) die Übersetzung der Symbolfolge S bedeute:

```
T("+" term)          = T(term)
T("-" term)          = T(term) "-"
T(term1 "+" term2)   = T(term1) T(term2) "+"
T(term1 "-" term2)   = T(term1) T(term2) "-"
T(factor1 "*" factor2) = T(factor1) T(factor2) "*"
```

```
T(factor1 "/" factor2) = T(factor1) T(factor2) "/"
T("("expression")")  = T(expression)
```

Damit reduziert sich die Aufgabe der Prozeduren zur Behandlung der Ausdrücke auf das Verzögern der Übermittlung der Operatoren. Der Leser sollte sich an dieser Stelle davon überzeugen, dass die Syntax die üblichen Prioritätsregeln von arithmetischen Operatoren korrekt widerspiegelt.

Die Übersetzungsregeln für die Wertzuweisung und die beiden Anweisungen zur Ein- und Ausgabe sind:

```
T(ident ":=" expression) = T(expression) STO ident
T("?" ident) = READ STO ident
T("!" expression) = T(expression) WRITE
```

Die Übersetzung von bedingten und repetierten Anweisungen ist nur unwesentlich komplizierter. Sie erfordert die Erzeugung von Sprungbefehlen. In gewissen Fällen ist die Zieladresse dieser Sprünge zur Zeit der Ausgabe dieser Befehle noch nicht bekannt. Es betrifft dies alle Sprünge nach vorwärts. Beharrt man strikte auf dem Prinzip der sequentiellen Übersetzung des Textes, dann muss in diesen Fällen der Text zweimal durchlesen werden. Diese weitverbreitete Strategie wird *Zweiphasen-Compilation* genannt. Die zweite Phase der Übersetzung besteht dann im Einsetzen der nunmehr bekannten Zieladressen. Wir wählen aber hier eine andere Lösung, die darin besteht, die Befehle in einem nachträglich beliebig zugreifbaren Array einzutragen, so dass die Zieladressen eingesetzt werden können, sobald sie bekannt sind. Dieses Flicken der unvollständigen Befehle wird im Jargon als *fixup* bezeichnet, und der Generator-Modul liefert die dazu nötige Prozedur. Die einzige Aufgabe neben der Ausgabe des Sprunges ist dann das Festhalten seines Index im Befehlsspeicher. Dieser Index gibt dann an, wo der Flick vorzunehmen ist. Die Einzelheiten sind aus Programm 8 ersichtlich, und zwar in den Prozeduren, die die if- und while-Anweisungen behandeln. Wir stellen ihre Aufgabe formal durch folgende zwei Regeln dar:

```
T("IF" condition "THEN" statement)  =
        T(condition)
        JPC L0
        T(statement)
    L0: ...
```

```
T("WHILE" condition "DO" statement) =
    L0: T(condition)
        JPC L1
        T(statement)
        JMP L0
    L1: ...
```

Als Beispiel sei nachfolgend die Befehlsfolge angegeben, die sich aus der Compilation der Prozedur zur Multiplikation zweier Zahlen (siehe Kapitel 7) ergibt. Die Kommentare auf der rechten Seite sind hier lediglich zur Information des Lesers angefügt.

```
 2   INT  0,5   allocate space
 3   LOD  1,3   x
 4   STO  0,3   a
 5   LOD  1,4   y
 6   STO  0,4   b
 7   LIT  0,0   0
 8   STO  1,5   z
 9   LOD  0,4   b
10   LIT  0,0   0
11   OPR  0,12  >
12   JPC  0,29
13   LOD  0,4   b
14   OPR  0,7   odd
15   JPC  0,20
16   LOD  1,5   z
17   LOD  0,3   a
18   OPR  0,2   +
19   STO  1,5   z
20   LIT  0,2   2
21   LOD  0,3   a
22   OPR  0,4   *
23   STO  0,3   a
24   LOD  0,4   b
25   LIT  0,2   2
26   OPR  0,5   /
27   STO  0,4   b
28   JMP  0,9
29   OPR  0,0   return
```

PL/0 Code für die Multiplikationsprozedur aus Kapitel 7.

```
IMPLEMENTATION MODULE PL0Parser;
  FROM SYSTEM IMPORT TSIZE;
  FROM Heap IMPORT ALLOCATE, ResetHeap;
  FROM TextWindows IMPORT Window, OpenTextWindow, Write,
    WriteLn, WriteCard, WriteString, Invert, CloseTextWindow;
  FROM PL0Scanner IMPORT
       Symbol, sym, id, num, Diff, KeepId, GetSym, Mark;
  FROM PL0Generator IMPORT Label, Gen, fixup;

  TYPE ObjectClass = (Const, Var, Proc, Header);
       ObjPtr = POINTER TO Object;
       Object = RECORD name: CARDINAL;
                  next:  ObjPtr;
                  CASE kind: ObjectClass OF
                    Const:  val: INTEGER |
                    Var:    vlev, vadr: CARDINAL |
                    Proc:   plev, padr, size: CARDINAL |
                    Header: last, down: ObjPtr
                  END
                END ;

  VAR topScope, bottom, undef: ObjPtr;
      curlev: CARDINAL;
      win: Window;

  PROCEDURE err(n: CARDINAL);
  BEGIN noerr := FALSE; Mark(n); Invert(win, TRUE);
    WriteCard(win, n, 1); Invert(win, FALSE)
  END err;

  PROCEDURE test(s: Symbol; n: CARDINAL);
  BEGIN
    IF sym < s THEN err(n);
      REPEAT GetSym UNTIL sym >= s
    END
  END test;

  PROCEDURE NewObj(k: ObjectClass): ObjPtr;
    VAR obj: ObjPtr;
```

```
BEGIN (*check for multiple definition*)
  obj := topScope↑.next;
  WHILE obj # NIL DO
    IF Diff(id, obj↑.name) = 0 THEN err(25) END ;
    obj := obj↑.next
  END ;
  (*now enter new object into list*)
  ALLOCATE(obj, TSIZE(Object));
  WITH obj↑ DO
    name := id; kind := k; next := NIL
  END ;
  KeepId; topScope↑.last↑.next := obj; topScope↑.last := obj;
  RETURN obj
END NewObj;

PROCEDURE find(id: CARDINAL): ObjPtr;
  VAR hd, obj: ObjPtr;
BEGIN hd := topScope;
  WHILE hd # NIL DO
    obj := hd↑.next;
    WHILE obj # NIL DO
      IF Diff(id, obj↑.name) = 0 THEN RETURN obj
        ELSE obj := obj↑.next
      END
    END ;
    hd := hd↑.down
  END ;
  err(11); RETURN undef
END find;

PROCEDURE expression;
  VAR addop: Symbol;

  PROCEDURE factor;
    VAR obj: ObjPtr;
  BEGIN WriteString(win, "factor"); WriteLn(win);
    test(lparen, 6);
    IF sym = ident THEN
      obj := find(id);
      WITH obj↑ DO
        CASE kind OF
```

```
          Const: Gen(0, 0, val) |
          Var:   Gen(2, curlev-vlev, vadr) |
          Proc:  err(21)
        END
      END ;
      GetSym
    ELSIF  sym = number  THEN
      Gen(0, 0, num); GetSym
    ELSIF sym = lparen THEN
      GetSym; expression;
      IF sym = rparen THEN GetSym ELSE err(7) END
    ELSE err(8)
    END
  END factor;

  PROCEDURE term;
    VAR mulop: Symbol;
  BEGIN WriteString(win, "term"); WriteLn(win);
    factor;
    WHILE (times <= sym) & (sym <= div) DO
      mulop := sym; GetSym; factor;
      IF mulop = times THEN Gen(1,0,4)
      ELSE Gen(1,0,5)
      END
    END
  END term;

BEGIN WriteString(win, "expression"); WriteLn(win);
  IF (plus <= sym) & (sym <= minus) THEN
    addop := sym; GetSym; term;
    IF addop = minus THEN Gen(1,0,1) END
  ELSE term
  END ;
  WHILE (plus <= sym) & (sym <= minus) DO
    addop := sym; GetSym; term;
    IF addop = plus THEN Gen(1,0,2) ELSE Gen(1,0,3) END
  END
END expression;

PROCEDURE condition;
  VAR relop: Symbol;
```

```
BEGIN WriteString(win, "condition"); WriteLn(win);
  IF sym = odd THEN
    GetSym; expression; Gen(1,0,6)
  ELSE
    expression;
    IF (eql <= sym) & (sym <= geq) THEN
      relop := sym; GetSym; expression;
      CASE relop OF
        eql: Gen(1,0, 8) |
        neq: Gen(1,0, 9) |
        lss: Gen(1,0,10) |
        geq: Gen(1,0,11) |
        gtr: Gen(1,0,12) |
        leq: Gen(1,0,13)
      END
    ELSE err(20)
    END
  END
END condition;

PROCEDURE statement;
  VAR obj: ObjPtr; L0, L1: CARDINAL;
BEGIN WriteString(win, "statement"); WriteLn(win);
  test(ident, 10);
  IF sym = ident THEN
    obj := find(id);
    IF obj↑.kind # Var THEN err(12); obj := NIL END ;
    GetSym;
    IF sym = becomes THEN GetSym
    ELSIF sym = eql THEN err(13); GetSym
    ELSE err(13)
    END ;
    expression;
    IF obj # NIL THEN
      Gen(3, curlev - obj↑.vlev, obj↑.vadr)  (*store*)
    END
  ELSIF sym = call THEN
    GetSym;
    IF sym = ident THEN
      obj := find(id);
      IF obj↑.kind = Proc THEN
```

```
        Gen(4, curlev - objt.plev, objt.padr)    (*call*)
      ELSE err(15)
      END ;
      GetSym
    ELSE err(14)
    END
  ELSIF sym = begin THEN GetSym;
    LOOP statement;
      IF sym = semicolon THEN GetSym
      ELSIF sym = end THEN GetSym; EXIT
      ELSIF sym < const THEN err(17)
      ELSE err(17); EXIT
      END
    END
  ELSIF sym = if THEN
    GetSym; condition;
    IF sym = then THEN GetSym ELSE err(16) END ;
    L0 := Label(); Gen(7,0,0); statement; fixup(L0)
  ELSIF sym = while THEN L0 := Label();
    GetSym; condition; L1 := Label(); Gen(7,0,0);
    IF sym = do THEN GetSym ELSE err(18) END ;
    statement; Gen(6,0,L0); fixup(L1)
  ELSIF sym = read THEN
    GetSym;
    IF sym = ident THEN
      obj := find(id);
      IF objt.kind = Var THEN
        Gen(1,0,14); Gen(3, curlev - objt.vlev, objt.vadr)
      ELSE err(12)
      END
    ELSE err(14)
    END ;
    GetSym
  ELSIF sym = write THEN
    GetSym; expression; Gen(1,0,15)
  END ;
  test(ident, 19)
END statement;

PROCEDURE block;
  VAR adr: CARDINAL;     (*data address*)
```

```
    L0: CARDINAL;      (*initial code index*)
    hd, obj: ObjPtr;

  PROCEDURE ConstDeclaration;
    VAR obj: ObjPtr;
  BEGIN WriteString(win, "ConstDeclaration"); WriteLn(win);
    IF sym = ident THEN
      GetSym;
      IF (sym = eql) OR (sym = becomes) THEN
        IF sym = becomes THEN err(1) END ;
        GetSym;
        IF sym = number THEN
          obj := NewObj(Const); obj↑.val := num; GetSym
        ELSE err(2)
        END
      ELSE err(3)
      END
    ELSE err(4)
    END
  END ConstDeclaration;

  PROCEDURE VarDeclaration;
    VAR obj: ObjPtr;
  BEGIN WriteString(win, "VarDeclaration"); WriteLn(win);
    IF sym = ident THEN
      obj := NewObj(Var); GetSym;
      obj↑.vlev := curlev; obj↑.vadr := adr; adr := adr+1
    ELSE err(4)
    END
  END VarDeclaration;

BEGIN WriteString(win, "block"); WriteLn(win);
  curlev := curlev + 1; adr := 3;
  ALLOCATE(hd, TSIZE(Object));
  WITH hd↑ DO
    kind := Header; next := NIL; last := hd;
    name := 0; down := topScope
  END ;
  topScope := hd; L0 := Label(); Gen(6,0,0);  (*jump*)
  IF sym = const THEN GetSym;
    LOOP ConstDeclaration;
```

```
      IF sym = comma THEN GetSym
      ELSIF sym = semicolon THEN GetSym; EXIT
      ELSIF sym = ident THEN err(5)
      ELSE err(5); EXIT
      END
    END
  END ;
  IF sym = var THEN GetSym;
    LOOP VarDeclaration;
      IF sym = comma THEN GetSym
      ELSIF sym = semicolon THEN GetSym; EXIT
      ELSIF sym = ident THEN err(5)
      ELSE err(5); EXIT
      END
    END
  END ;
  WHILE sym = procedure DO
    GetSym;
    IF sym = ident THEN GetSym ELSE err(4) END ;
    obj := NewObj(Proc);
    obj↑.plev := curlev; obj↑.padr := Label();
    IF sym = semicolon THEN GetSym ELSE err(5) END ;
    block;
    IF sym = semicolon THEN GetSym ELSE err(5) END
  END ;
  fixup(L0); Gen(5,0,adr); (*enter*)
  statement;
  Gen(1,0,0);  (*return*)
  topScope := topScope↑.down; curlev := curlev - 1
END block;

PROCEDURE Parse;
BEGIN noerr := TRUE; topScope := NIL; curlev := 0;
  Write(win, 14C); ResetHeap(bottom);
  GetSym; block;
  IF sym # period THEN err(9) END
END Parse;

PROCEDURE EndParser;
BEGIN CloseTextWindow(win)
END EndParser;
```

```
BEGIN ALLOCATE(undef, TSIZE(Object)); ALLOCATE(bottom, 0);
  WITH undef↑ DO
    name := 0; next := NIL; kind := Var; vlev := 0; vadr := 0
  END ;
  OpenTextWindow(win, 0, 66, 234, 508, "PARSE");
END PL0Parser.
```

Programm 8: PL/0 Parser

Die Struktur des kompletten Compiler-Interpreter-Systems ist aus Fig. 8 ersichtlich. Darin deuten die Pfeile die Richtung von Importen von Objekten aus den Modulen an. Das System von Modulen wird beherrscht von einem Hauptmodul PL0, welcher zuerst den Namen der Quelldatei verlangt und danach den Compiler und, falls keine Fehler festgestellt wurden, den Interpreter zur Ausführung des Programmes aufruft. Die Figuren 9 und 10 schliesslich zeigen Bildschirmdaten, die bei der Compilation und Ausführung von zwei kurzen Programmen entstehen.

```
MODULE PL0;  (*NW WS 83/84*)
  FROM Terminal IMPORT Read;
  FROM FileSystem IMPORT Lookup, Response, Close;
  FROM TextWindows IMPORT Window,
       OpenTextWindow, Write, WriteLn, WriteString,
CloseTextWindow;
  FROM PL0Scanner IMPORT InitScanner, source, CloseScanner;
  FROM PL0Parser IMPORT Parse, noerr, EndParser;
  FROM PL0Generator IMPORT InitGenerator, EndGenerator;
  FROM PL0Interpreter IMPORT Interpret, EndInterpreter;

  CONST NL = 27;  (*max file name length*)

  VAR ch: CHAR;
      win: Window;
      FileName: ARRAY [0..NL] OF CHAR;

  PROCEDURE ReadName;
    CONST DEL = 177C;
    VAR i: CARDINAL;
  BEGIN Read(ch); FileName := "DK.";
    i := 3;
```

```
    WHILE (CAP(ch) >= "A") & (CAP(ch) <= "Z")
       OR (ch >= "0") & (ch <= "9")
       OR (ch = ".") OR (ch = DEL) DO
      IF ch = DEL THEN
        IF i > 3 THEN Write(win, DEL); i := i-1 END
      ELSIF i < NL THEN
        Write(win, ch); FileName[i] := ch; i := i+1
      END ;
      Read(ch)
    END ;
    IF (3 < i) & (i < NL) & (FileName[i-1] = ".") THEN
      FileName[i] := "P"; i := i+1;
      FileName[i] := "L"; i := i+1;
      FileName[i] := "0"; i := i+1; WriteString(win, "PL0")
    END ;
    FileName[i] := 0C
  END ReadName;

BEGIN OpenTextWindow(win, 0, 0, 704, 66, "DIALOG");
   LOOP WriteString(win, "in> "); ReadName;
    IF ch = 33C THEN EXIT END ;
    Lookup(source, FileName, FALSE);
    IF source.res = done THEN
      InitScanner; InitGenerator; Parse; Close(source);
      IF noerr THEN
        WriteString(win, " interpreting"); Interpret
      ELSE WriteString(win, " incorrect")
      END
    ELSE WriteString(win, " not found")
    END ;
    WriteLn(win)
  END ;
  CloseScanner; EndParser; EndGenerator; EndInterpreter;
  CloseTextWindow(win)
END PL0.
```

Programm 9: Hauptmodul des Compiler-Interpreter Systems

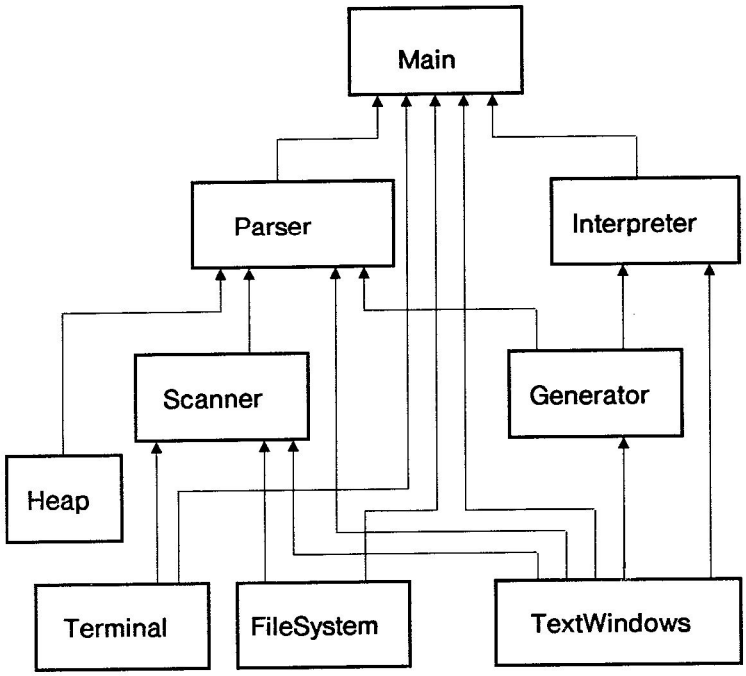


Fig. 8. Modulstruktur des PL/0 Systems

Fig. 8: Modulstruktur des PL/0 Systems

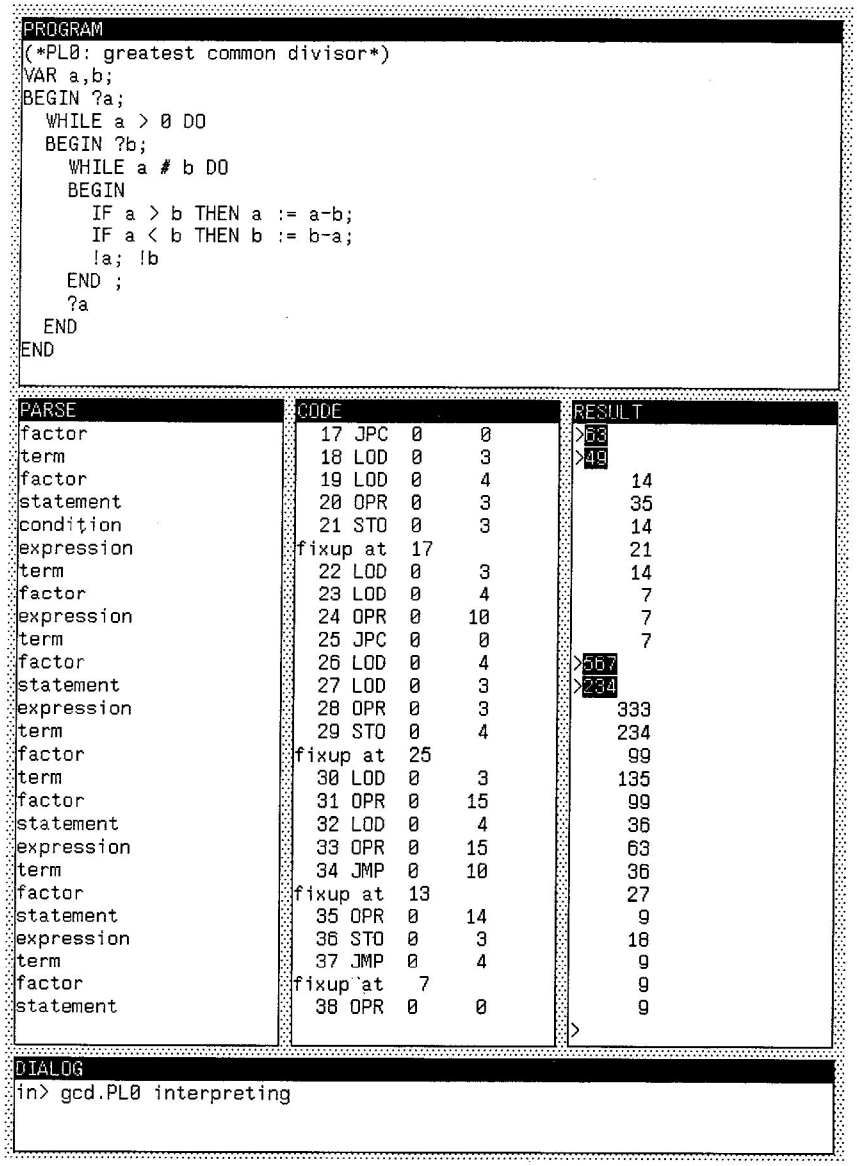


Fig. 9. Beispielprogramm: Grösster gemeinsamer Teiler

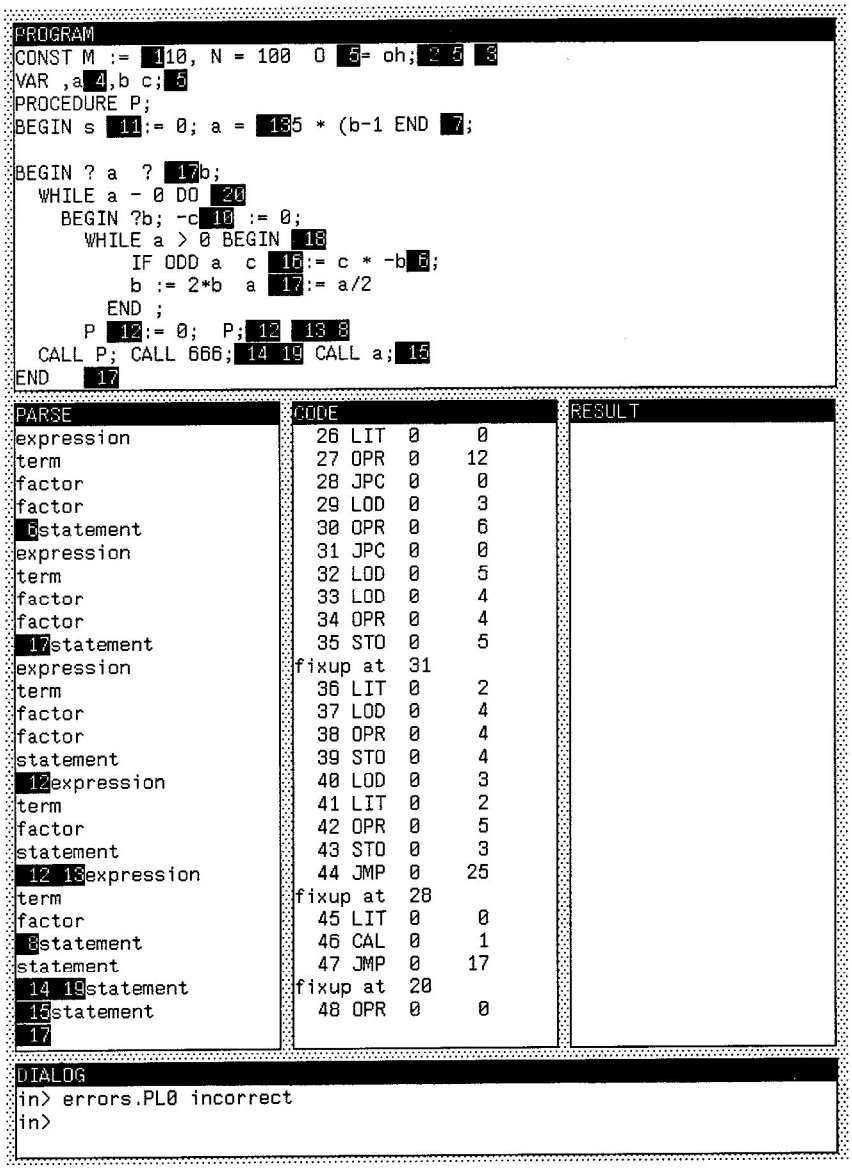


Fig. 10. Beispielprogramm mit syntaktischen Fehlern

12. Eine Spracherweiterung: Prozesse

Programme ganz allgemein sind Beschreibungen von Automaten, die sich strikte nach einer vorgeschriebenen Regel verhalten. Wir betrachten den Computer als Werkzeug, das sich je nach Programm in einen anderen Automaten verwandeln lässt. Im Lichte dieser Betrachtungsweise erscheint das Programmieren als ein Konstruieren von Maschinen aus Bauteilen, die die verwendete Programmiersprache zur Verfügung stellt. Nun ist es in keinem anderen Zweig des Ingenieurwesens derart leicht, einmal konstruierte Maschinen abzuändern, zu erweitern, veränderten Bedürfnissen anzupassen. Dies hat zur Folge, dass sich der Programmierer wie kaum ein anderer Ingenieur auf die leichte, scheinbar kostenlose Wandelbarkeit seiner Produkte verlässt. So gibt es in der Praxis denn auch eher selten programmierte Systeme, die je den Zustand der Endgültigkeit erreichen. Besonders bei der Konstruktion komplexer Systeme ist es daher wichtig, sie so zu planen, dass ein Ausbau in verschiedenster Art möglich ist. Besondere Aufmerksamkeit ist dabei der Modularisierung, d.h. der Aufteilung in einzelne Module zu schenken. Ist eine Grundstruktur glücklich gewählt, so lassen sich Änderungen auf wenige Orte lokalisieren, was für die Übersichtlichkeit und Zuverlässigkeit enorm wichtig ist.

Der Charakter der ständigen Evolution ist aber nicht nur Programmen eigen, sondern auch den Programmiersprachen. Dies erklärt sich leicht aus dem Umstand, dass deren Übersetzer auch wieder wandelbare Programme sind. Dieses Kapitel sei daher einer solchen Spracherweiterung und deren Realisierung gewidmet. Sie ist aber lediglich ein Beispiel unter vielen für das viel allgemeinere Phänomen eines Systemausbaus. Die vorerst formulierten Regeln des Vorgehens haben denn auch Gültigkeit über das nachher verwendete Beispiel hinaus. Weitere, ähnlich Projekte finden sich unter den Aufgaben am Ende dieses Buches.

Mit grossem Vorteil wird bei einer beabsichtigten Erweiterung einer Programmiersprache die Aufmerksamkeit zuerst den konzeptionellen Aspekten gewidmet. Die neuen Sprachelemente sollen in einer allgemeingültigen Weise erfasst und definiert werden. Eine Ausrichtung auf eine bestimmte Art ihrer Mechanisierung ist wenn möglich zu vermeiden. Selbstverständlich kennt der erfahrene Konstrukteur gewisse Techniken der Implementation, und er weiss sogar deren Vor- und Nachteile einigermassen abzuschätzen. Die Kunst liegt

aber vor allem darin, sich nicht allzu früh auf eine bestimmte Lösung festzulegen oder auszurichten. Wir stellen daher einen Plan auf, der angeben soll, worauf sich unsere Hauptaufmerksamkeit in zeitlicher Folge richten soll.

1. Definition der Grundkonzepte, die neu eingeführt werden sollen.
2. Definition einer Darstellung der neuen Sprachelemente. Formale Syntax und Semantik.
3. Festlegung der für die neuen Sprachelemente zu erzeugenden Befehlsfolgen. Falls nötig, Definition der Erweiterungen der interpretierenden Maschine.
4. Bestimmung der Datentypen und Datenstrukturen im Compiler, die zur Darstellung der neuen Sprachelemente notwendig sind.
5. Programmierung der Algorithmen, welche die neuen Sprachelemente verarbeiten. Integration der neuen Compilerteile.
6. Prüfung der Korrektheit der Implementation anhand von Testprogrammen mit und ohne neue Sprachelemente.

Wir durchlaufen nun diese Schritte mit Ausnahme der zwei letzten für einen beispielhaften Fall. PL/0 soll von einer rein sequentiellen in eine Sprache erweitert werden, in der gleichzeitige Programmabläufe beschrieben werden können.

Schritt 1: Grundkonzepte.

Wir weichen vom Axiom des rein sequentiellen Ablaufs von Programmen ab, indem wir die Vereinbarung von mehreren, in sich selber sequentiellen Prozessen zulassen, die, einmal gestartet, gleichzeitig ablaufen dürfen. Dieses Konzept ist aber nur dann von Interesse, wenn die Prozesse irgendwie miteinander kommunizieren können. Die einfachste Art dazu ergibt sich aus der Verwendung gemeinsamer Variablen. Damit lassen sich Prozesse auch *synchronisieren*, d.h. so programmieren, dass sie an bestimmten Punkten aufeinander warten. Nehmen wir z.B. an, dass ein Prozess P nach der Anweisung A auf die Beendigung der Anweisung B durch einen Prozess Q warten soll, so kann dies programmiert werden, indem eine gemeinsame Variable w wie folgt verwendet wird. w habe anfänglich den Wert TRUE.

```
P: ...    A; WHILE w DO END ; ...
```

Q: ... B; w := FALSE; ...

Auch wenn wir zulassen, dass solche Synchronisationen selten ausgeführt werden - wir sprechen in diesem Fall von *lose gekoppelten* Prozessen - so hat doch diese Technik der Synchronisierung einen entscheidenden Nachteil. Sie legt es nahe, jedem Prozess einen eigenen Prozessor zuzuordnen. Weshalb? Eine Konzentration der gesamten Arbeit auf wenige oder sogar einen einzigen Prozessor ist unrealistisch, weil die Prozesse sogar beschäftigt bleiben, wenn sie lediglich warten. Die Anweisung "WHILE w DO END" verkörpert eine perverse Art des geschäftigen Wartens (*busy waiting*) und verschleiert die wirkliche Absicht des Programmierers. Sie ist gewählt, weil kein zutreffendes Sprachelement vorhanden ist. Wir korrigieren diesen Missstand, indem wir eine neue Anweisung für das Warten einführen.

Dies bedingt hinwiederum die Einführung einer neuen Art von Objekten, nämlich von Zielen, auf die gewartet wird. Je nach Betrachtungsweise werden diese Ziele Bedingungen (conditions), auf deren Erfüllung gewartet wird, oder Ereignisse (events), auf deren Eintreffen gewartet wird, genannt. Weniger sinnvoll ist dafür die in der Literatur ebenfalls verbreitete Bezeichnung Warteschlange (*queue*), da sie auf eine mögliche Implementation hindeutet. Wir wählen hier die neutrale Bezeichnung *Signal:* ein Prozess wartet auf das Eintreffen eines Signals, das von einem anderen Prozess ausgesendet wird. Damit ist auch angedeutet, dass *zwei* Operationen mit Signalen benötigt werden, nämlich das *Senden* eines Signals sowie das *Empfangen* eines Signals, dem unter Umständen eine Wartezeit vorangeht, bis das Signal eintrifft.

Die Synchronisation von Prozessen mit diesen neuen Objekten anstatt mit konventionellen Variablen hat den grossen Vorteil, dass Prozesse einander nicht mit Namen kennen müssen. Sie warten auf ein Signal, das unabhängig von der Identität des Absenders einen gewissen, zur Fortsetzung notwendigen Zustand des Systems ankündigt. Dieser Umstand ist bei Simulationen von Systemen mit diskreten Ereignissen besonders vorteilhaft. Die hier geplante Erweiterung erschliesst damit ein vielseitiges neues Anwendungsgebiet.

Entscheidend ist nun aber der Umstand, dass es durch das explizite Konzept der Signale möglich wird, ein System mit mehreren Prozessen durch einen Computer mit einem einzigen Prozessor zu implementieren. Die Gleichzeitigkeit wird dabei allerdings nur simuliert. Falls jedoch ein Prozess in den Wartezustand gelangt, so kann der Prozessor unverzüglich zur Ausführung anderer Prozesse eingesetzt werden. Es ist nicht nötig, stets zu prüfen, ob das Warten abgebrochen

werden kann, denn die einzige Art, wie dies zustande kommen kann, ist durch das Absenden des betreffenden Signals durch den ausgeführten Prozess. Wartende Prozesse können daher unbeaufsichtigt gelassen werden und beanspruchen keine Prozessorzeit.

Grundsäztlich wäre es nötig, neben den Prozessen und Signalen noch ein weiteres Konzept neu einzuführen, um simultane Prozesse nützlich einsetzen zukönnen. Es ist dies der gegenseitige Ausschluss von Prozessen aus bestimmten, kritischen Abschnitten von Programmen. Weil wir uns hier mit der Simulation von gleichzeitigen Prozessen mittels eines einzigen Prozessors befassen, kann das Problem des gegenseitigen Ausschlusses leicht dadurch gelöst werden, dass innerhalb von kritischen Abschnitten keine Anweisungen plaziert werden, die den Prozessor von einem Prozess auf einen andern umleiten könnten. Wir können daher darauf verzichten, zur Bezeichnung eines kritischen Abschnittes eine spezielle Sprachstruktur einzuführen.

Schritt 2: Formale Definition von Syntax und Semantik.

Wir beschränken uns darauf, die erweiterte Syntax als EBNF Produktionen anzugeben. Die Änderungen betreffen nur die Strukturen ***block*** und ***statement.*** Das Senden eines Signals s werde mit !s, das Erwarten und Empfangen durch ?s ausgedrückt. Die Verwendung derselben Schreibweise wie für die Aus- und Eingabe von Daten deutet die enge Verwandtschft der beiden Konzepte an.

```
block =   ["CONST" ident "=" number {"," ident "=" number} ";"]
          ["VAR" ident {"," ident} ";"]
          ["SIGNAL" ident {"," ident} ";"]
          {("PROCEDURE" | "PROCESS") ident ";" block ":"} statement.

statement = [ "!" ident | "?" ident |
          ident ":=" expression | "CALL" ident | ... ].
```

Prozess-Vereinbarungen besitzen dieselbe Form wie Prozedur-Vereinbarungen. Falls in einem Aufruf (call) ein Prozess-Bezeichner verwendet wird, so bedeutet diese Anweisung die Erzeugung eines neuen Prozesses, der simultan (oder quasi-simultan) mit dem aufrufenden Prozess ausgeführt werden soll. Dies steht im Gegensatz zum Aufruf einer Prozedur, wo der aufrufende Prozess erst wieder aufgenommen wird, wenn die Prozedur beendet ist. Die Sende- und Empfangs-Anweisungen beziehen sich natürlich auf ein Signal. Die Bedeutung einer Signalübermittlung liegt darin, dass dem Empfänger mitgeteilt wird, dass

das System (von Prozessen) einen bestimmten Zustand erreicht hat. Diesen eigentlichen Informationsgehalt eines Signals s wollen wir durch das Prädikat P_s bezeichnen. Die Semantik der beiden neuen Anweisungen kann jetzt mittels einer *Vorbedingung* zum Senden und einer *Konsequenz* des Empfangs eines Signals axiomatisch definiert werden.

$$\{P_s\}\ !s \qquad\qquad ?s\ \{P_s\}$$

Schritt 3: Erweiterung der PL/0 Maschine und Definition des erzeugten Codes.

Es ist offensichtlich, dass die wesentliche konzeptionelle Erweiterung von PL/0 durch simultane Prozesse auch einen entsprechenden Ausbau der PL/0 Maschine erfordert. Wir gehen im Folgenden ausdrücklich von der Annahme aus, dass die Gleichzeitigkeit nicht nur simuliert werden *kann*, sondern sogar simuliert werden *muss*. Die PL/0 Maschine bleibt daher weiterhin als sequentielles Programm beschreibbar. Wir wenden uns aber vorerst nicht der Frage zu, wie ein einziger Prozessor mehrere Prozesse bedienen soll, sondern erwähnen zuerst die bei der Erzeugung eines Prozesses auftretenden Probleme. Sofort stellt sich dabei die Problematik der Speicherverwaltung in den Vordergrund. Wenn mehrere Prozesse gleichzeitig existieren und keine einfache Beziehung zwischen den zeitlichen Folgen ihrer Entstehung und Auflösung besteht, so kann die bewährte Stapelstrategie nicht aufrecht erhalten werden. Wir wollen allerdings hier nicht auf allgemeine Methoden der Speicherverwaltung eingehen und postulieren daher einige vereinfachende Regeln ohne jeglichen Anspruch auf effiziente Verwendbarkeit in praktischen Systemen.

1. Prozesse werden nur im Hauptprogramm vereinbart. Dadurch wird vermieden, dass ein Prozess Q Bezug auf lokale Variablen eines Prozesses P nehmen kann, in dem Q selbst lokal ist, nachdem sich P bereits aufgelöst hat.

2. Bei jeder Erzeugung eines neuen Prozesses wird ihm ein Speicherplatz fester Grösse zugeordnet. Darin befindet sich sein eigener Stapel (*workspace*).

3. Am Ende eines Prozesses wird der Speicher wieder an die Verwaltung zurückgegeben. Ist diese Verwaltung geschickt organisiert, so kann er bei einer späteren Prozesserzeugung wieder verwendet werden.

Der Hauptnachteil dieser primitiven Strategie liegt darin, dass der Stapel in einem festen Speicherplatz untergebracht werden muss, obwohl die grösste Ausdehnung eines Stapels nicht im voraus bekannt ist. Ihr Vorteil liegt aber in

ihrer Einfachheit; insbesondere kann der Speicher weiterhin als Array dargestellt werden, und die einzelnen Stapel sind zusammenhängende Abschnitte dieses Arrays. Wir teilen nun den Datenspeicher S in Abschnitte ein, von denen jeder einem Prozess gehört, und deren erste Elemente einen Prozess-Descriptor bilden. Den Abschnitt nennen wir ein *Prozess-Segment*. Jedes Prozess-Segment besteht aus einzelnen Prozedur-Segmenten und ist als Stapel verwaltet. Die einzelnen Prozess-Segmente sind nach Fig. 11 durch Zeiger (Indices) verkettet. Die Kette aller aktiven Segmente bildet den sogenannten *Prozess-Ring*. Das neue Register CP (current process) bezeichnet denjenigen Prozess, der sich gegenwärtig in Ausführung befindet.

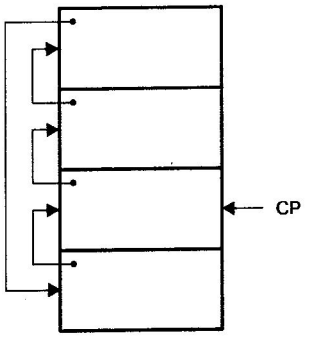


Fig. 11. Datenspeicher mit Prozess-Segmenten.

Zu beachten ist ferner, dass das Hauptprogramm selbst am einfachsten auch als Prozess betrachtet wird. Es ist vernünftig, ihm einen grösseren, der Anzahl globaler Variablen entsprechenden Speicherplatz zur Verfügung zu stellen.

Die Register P, T und B (siehe Kapitel 10) enthalten die entsprechenden Werte des gegenwärtig ausgeführten Prozesses. Die Registerwerte aller anderen Prozesse müssen im Segment abgespeichert werden. Das Schema eines Segmentes mit den entsprechenden Datenfeldern ist in Fig. 12 angegeben.

SL bezeichnet nach wie vor die statische Kette der Prozedur-Segmente (siehe Kapitel 10), *link* bezieht sich auf den Prozess-Ring, und *state* gibt den Zustand eines Prozesses an. Er ist entweder bereit (state = 0) oder wartet auf ein Signal (state > 0). Es liegt eigentlich nahe, dem wartenden Zustand als weiteres Attribut einen Verweis auf das erwartete Signal beizuordnen. Dies ist jedoch darum wenig sinnvoll, weil der wartende Prozess selbst gar nie in die Lage kommt, auf dieses

Signal Bezug zu nehmen. Vielmehr ist es zweckmässig, umgekehrt dem Signal einen Verweis auf den Prozess zuzuordnen. Weil mehrere Prozesse auf ein und dasselbe Signal warten können, ergibt sich im allgemeinen Fall eine Verkettung dieser Prozesse als geeignete Lösung. Der Anfang dieser Warteschlange von Prozessen bildet gleichsam den "Wert", der dem Signal zugeordnet ist. Jedem vereinbarten Signal wird daher vom Compiler wie einer Variablen ein Speicherplatz zugewiesen. Für den Programmierer hat das Signal jedoch keinen Wert, der gelesen und durch eine Zuweisung verändert werden könnte. Das Feld Q im Prozess-Descriptor dient zur Speicherung dieses Kettengliedes (queue). Der letzte Prozess in der Kette erhält den Zeigerwert Q = NIL.

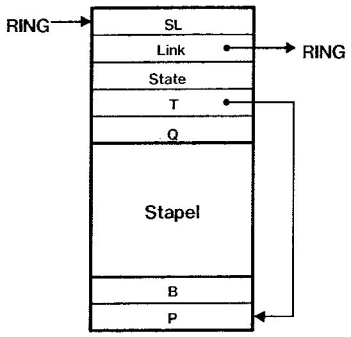


Fig. 12. Segment eines einzelnen Prozesses.

Nachdem nun die sogenannte Architektur oder Struktur der neuen PL/0 Maschine feststeht, müssen noch die neuen Befehle definiert werden. Es handelt sich um deren vier, nämlich

1. den Befehl STRT, um einen neuen Prozess zu erzeugen und zu starten,
2. den Befehl SEND, um ein Signal abzusenden,
3. den Befehl RECEIVE, um ein Signal zu empfangen,
4. den Befehl STOP, um einen Prozess aufzulösen.

Der Befehl STRT besitzt analog zu CAL die Anfangsadresse eines Prozesses als Parameter. SEND und RECEIVE besitzen die Datenadresse des Signals als Parameter, während STOP in der parameterlosen Klasse der Operatoren untergebracht werden kann.

Nachdem nun auch die Erweiterungen des Befehlssatzes definiert sind,

bleiben noch die Algorithmen der einzelnen Befehle zu beschreiben. Wir verzichten auf eine ausführliche Programmierung und beschränken uns auf einige wesentlich Punkte. Die Implementation der Befehle STRT und STOP ist sehr einfach, wenn nach dem erwähnten Prinzip der Speicherzuteilung vorgegangen wird. Wir wenden uns daher sogleich den Befehlen SEND und RECEIVE zu, indem wir die Zustände eines Prozess-Segmentes vor und nach der Befehlsausführung beschreiben.

Der Prozess wird in die Zeigerkette, die von s ausgeht und die Warteschlange der auf s wartenden Prozessen darstellt, eingereiht (s. Fig. 13). Nach der Suspendierung des Prozesses kann im Prinzip jeder beliebige Prozess, der den Zustand *bereit* aufweist, wieder aufgenommen werden. Da alle Prozesse im Ring verkettet sind, ist es naheliegend, entlang diesem Ring zu suchen und den ersten auszuwählen, der bereit ist. Existiert kein solcher Prozess, so befindet sich das System in einer Patt-Situation, wo jeder Prozess auf das Signal eines anderen wartet. Dieser Zustand wird als Verklemmung *(deadlock)* bezeichnet, und kommt in einem korrekt programmierten System nicht vor. Eine Fehlermeldung ist also in diesem Fall am Platz.

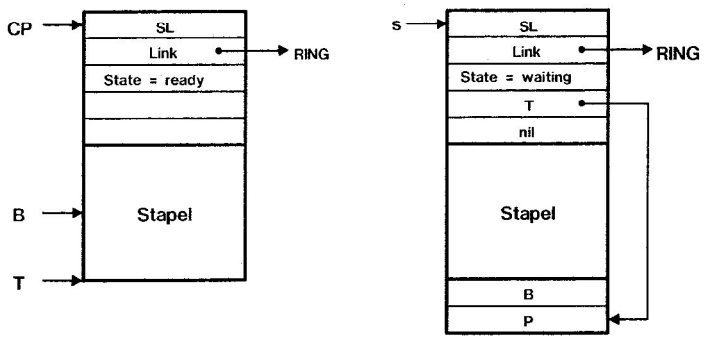


Fig. 13. Prozess-Segment vor und nach ?s

Bei der Implementation des Befehls SEND ist man geneigt, lediglich den ersten Prozess aus der Warteschlange herauszulösen und seinen Zustand als bereit zu markieren. Dies ist jedoch nicht zulässig, und zwar weil die für die Signaloperationen postulierten Axiome verlangen, dass der empfangende Prozess dieselbe Bedingung vorfindet, die der Absender signalisiert. Würde nun der Prozessor nicht sogleich vom Absender an den wartenden Empfänger übergeben,

so könnte ein dritter Prozess die signalisierte Bedingung inzwischen wieder invalidieren. Daraus ergibt sich, dass der Befehl SEND ganz ähnlich wie RECEIVE implementiert werden muss, wobei allerdings der Sender den Zustand *bereit* beibehält und der Empfänger durch den Befehl explizite bezeichnet ist. Er muss also nicht im Ring zuerst noch gesucht werden.

Es bleibt zuletzt noch der Fall zu behandeln, wo die Warteschlange des gesendeten Signals leer ist. Wie ist die Wirkung eines Signals zu definieren, auf das kein Prozess wartet? Es ist üblich, sie in diesem Fall einfach als nichtig zu erklären und mit dem sendenden Prozess weiterzufahren. Allerdings gibt es auch Gründe, diesen Fall als unbeabsichtigten Programmierfehler zu betrachten und dementsprechend zu interpretieren. Sinnvoll ist es zumindest, in diesem Fall den Prozessor freizugeben.

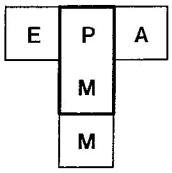
Schritt 4: Erweiterungen im Deklarationsteil des Compilers.

Die hier notwendigen Zusätze sind relativ beschränkt und einfach. Sicherlich wird der Wertebereich des Typs *symbol* erweitert, um die neuen Symbole zu repräsentieren (siehe Programm 4). Im weiteren erhält der Typ *object* die zwei neuen Werte *prozess* und *signal*. Der Typ *fct* wird um die vier neuen Befehle erweitert. Neu unter den Variablen im Interpreter finden wir das Prozess-Register CP, das zum Descriptor des gegenwärtig bearbeiteten Prozesses zeigt.

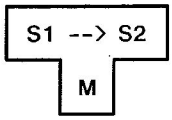
Wir überspringen eine ausführliche Programmierung der Erweiterungen des Compilers, die nicht aufwendig sind. In der Praxis wird diese Technik der Prozessverwaltung in Mehrprozess-Systemen (*time-sharing*) verwendet. Allerdings gelangen meistens raffiniertere Strategien der Speicherverwaltung und des Einsatzes des Prozessors (*process scheduling*) zur Anwendung. Weil das Thema dieses Kapitels jedoch die Erweiterung von Sprachen und deren Implementation ist, soll auf Verbesserungsmöglichkeiten des partikulären Beispiels nicht eingegangen werden.

13. Technik der Compilerentwicklung und -Übertragung

In den vorangehenden Kapiteln wurde ein PL/0 Compiler entwickelt und programmiert. Daraus wird ersichtlich, dass ein Compiler im allgemeinen ein komplexes Programm ist, und dass deshalb die Verwendung einer höheren Programmiersprache zur Implementierung besonders zu empfehlen ist. Selbstverständlich gelten auch die Regeln und Praktiken der systematischen Programmentwicklung genau wie für jeden anderen komplizierten Algorithmus. Es gibt aber doch einige Punkte, die für Compiler spezifisch sind und deshalb hier noch erwähnt werden sollen. Insbesondere beziehen sich diese auf den relativ häufigen Fall der Erweiterung. Da ein Compiler, ja selbst eine Programmiersprache selten von Anfang an in endgültiger Fassung in Angriff genommen wird, wird vielmehr ein Kern der Sprache implementiert, worauf das Endziel schrittweise angenähert wird. Auf diese Technik soll nachfolgend näher eingegangen werden, doch führen wir vorerst noch eine Notation zur Darstellung von Programmentwicklungen ein. Das Bild



soll andeuten, dass Eingabedaten E mittels eines Programms P, formuliert in der Sprache M und daher interpretierbar durch einen Computer M, in Ausgabedaten A verwandelt werden. Handelt es sich beim Programm P um einen Compiler, der Sätze aus der Sprache S1 in die Sprache S2 übersetzt, so wird dieser Compiler, der selbst in der Sprache M formuliert ist, durch die Figur



dargestellt. Diese Form eines T hat denn auch die Bezeichnung *T-Diagramme* geprägt. Mit ihrer Hilfe können wir den üblichen Vorgang einer Compilation mit

nachfolgender Ausführung eines Programmes P nach Fig. 14 schematisch beschreiben, wobei in beiden Schritten derselbe Computer M verwendet wird.

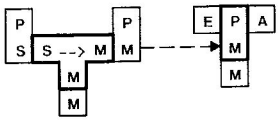


Fig.14. Schematische Darstellung von Compilation und Ausführung

In dieser Figur tritt nun die charakteristische Eigenschaft der T-Diagramme zutage: sie werden wie Dominosteine aneinander gereiht. Sich berührende Felder müssen dabei denselben Bezeichner tragen. Insbesondere wird deutlich, dass der Compiler das Programm P, formuliert in der Quellsprache S, in dasselbe Programm P, jedoch formuliert in der Zielsprache M, übersetzt. Als weiteres Beispiel einer Anwendung von T-Diagrammen zeigen wir den Vorgang einer Mehrphasen-Compilation in Fig. 15.

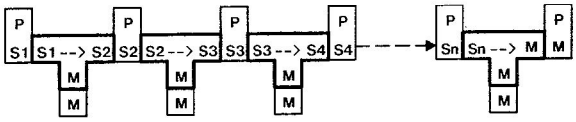


Fig. 15. n-Phasen Compilation eines Programms P

Der Vorgang einer interpretativen Ausführung eines Programms P, formuliert in einer Sprache (oder einem Code) H, durch einen am Computer M implementierten Interpreter, wird durch das Schema in Fig. 16 veranschaulicht. Seine Verallgemeinerung ist die mehrstufige Interpretation, ebenfalls dargestellt in Fig. 16. Ein Beispiel dafür wäre der PL/0 Code für H1, implementiert an einer mikroprogrammierten Maschine, deren Mikrocode H2 ist.

Nach dieser Einführung der T-Diagramme wenden wir uns nun der Technik der schrittweisen Compiler-Entwicklung zu. Hier ergeben sich nämlich beachtliche Vorteile, wenn ein Compiler für eine Quellsprache S selbst auch in S formuliert wird. Nehmen wir vorerst einmal an, dass ein Compiler für ein S existiere, und zwar sowohl in S selbst als auch in der Maschinensprache des zur Verfügung stehenden Computers M. Nun soll S in eine Version S' weiterentwickelt werden. Dies geschieht in zwei Schritten, nämlich (1) durch

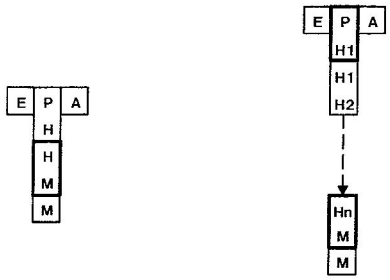


Fig. 16. Interpretative Ausführung eines Programms P

Programmierung der Erweiterungen in S (siehe Kapitel 12) und (2) durch Neuformulierung des Compilers, in der die neuen Sprachelemente nutzbringend verwendet werden. Der grosse Vorteil der Formulierung eines Compilers in seiner eigenen Quellsprache besteht also darin, dass der Compiler selbst von einer Erweiterung sofort profitieren kann. Deshalb kann die Entwicklung des Compilers mit einer relativ einfachen Sprache beginnen und durch Wiederholung dieser zwei Schritte zum Endprodukt emporgehoben werden. Der Fachausdruck für dieses sich selbst Hochziehen lautet *bootstrapping*. Ein solcher Akt wird in Fig. 17 gezeigt, und lässt sich beliebig oft wiederholen.

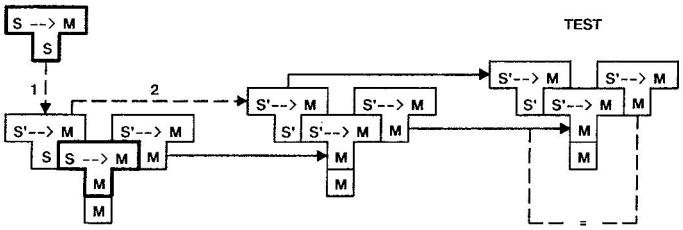


Fig. 17. Bootstrap eines Compilers von S nach S'

Erwähnenswert ist noch der dritte, in Fig. 17 abgebildete Schritt. Er dient zur Prüfung der Richtigkeit der Umprogrammierung des Compilers von S nach S'. Er erfolgt rein mechanisch, indem festgestellt wird, ob die Resultate der Schritte 2 und 3 genau identisch seien. Die Existenz eines solchen rigorosen Tests ist für die Praxis von unschätzbarem Wert.

Es stellt sich natürlich sofort die Frage nach dem Anfangsschritt. Wie wird ein Compiler für eine erste Version von S, formuliert in S, realisiert? Die übliche Lösung liegt in der Übersetzung "von Hand" in eine andere, auf M verfügbare Programmiersprache H. Es hat sich als vorteilhaft erwiesen, dafür ebenfalls eine höhere Sprache anstelle von Assembler-Code zu verwenden. Der Anfangsschritt ist in Fig. 18 gezeigt.

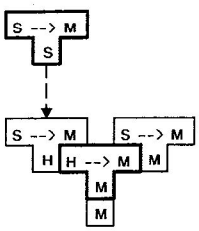


Fig. 18. Realisierung eines Compilers für S.

Der Anfangsschritt wird durch den glücklichen Umstand wesentlich erleichtert, dass die Programmierung in der "fremden" Hilfssprache H durchaus ohne Rücksicht auf Effizienz erfolgen darf. Die Ineffizienz des Initialproduktes pflanzt sich nämlich nicht fort und macht sich nur im ersten Bootstrap bemerkbar, wonach das Hilfsprodukt weggeworfen werden kann.

Tatsächlich braucht ein Bootstrap nicht unbedingt eine Spracherweiterung zu beinhalten. Er kann sich auch auf eine Verbesserung der Compilationsmethode und des erzeugten Code beziehen. Ein solcher Bootstrap wird in Fig. 19 veranschaulicht. In der Notation S $x \rightarrow y$ M bezeichne $x=a$ die Verwendung der alten, $x=b$ der verbesserten Compilationsmethode oder Code-Generierung, $y=a$, dass der Compiler selbst durch die alte, $y=b$ durch die neue Version des Compilers erzeugt wurde. Offensichtlich erlaubt also die Methode des Bootstrapping die anfängliche Verwendung einfacher, aber unter Umständen nicht optimaler Code-Erzeugung, und offeriert die Möglichkeit der nachträglichen Verbesserung. Ist der Compiler in der eigenen Quellsprache formuliert, so kommt er als erstes Programm in den Genuss dieser Verbesserungen.

Die angefügte Testphase erlaubt wiederum die Prüfung, ob die verbesserte Version des Compilers korrekt ist und sich selbst zu reproduzieren vermag.

Zum Abschluss sei noch auf den Problemkreis der *Compiler-Übertragung*

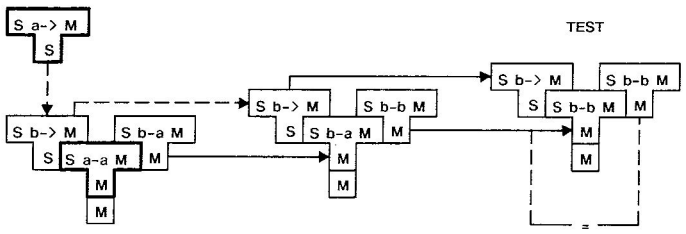


Fig. 19. Bootstrap zur Verbesserung eines Compilers.

eingegangen. Eine Sprache S soll also auf einem Computer M2 verfügbar gemacht werden, indem ein existierender Compiler von M1 auf M2 übertragen wird. Compiler stellen unter allen Programmen gerade in dieser Beziehung einen Sonderfall dar. Die Verwendung einer M1 und M2 gemeinsamen Sprache bringt nämlich noch keine Lösung. Ein Compiler erzeugt schliesslich Befehlsfolgen seiner Zielsprache und ist damit naturgemäss auf einen bestimmten Zielcomputer zugeschnitten und somit nicht übertragbar (portabel). Eine Übertragung erfordert daher immer das Erstellen eines prinzipiell neuen Programmes. Allerdings können wesentliche Teile wie z.B. die Syntaxanalyse und die Fehlerbehandlung unverändert übernommen werden, falls der Compiler zweckmässig organisiert ist (siehe Kapitel 8 und 9). Dennoch ist dieser wesentliche Schritt einer Umprogrammierung unumgänglich.

Die naheliegende Lösung, die verbleibenden Arbeiten auf ein Minimum zu reduzieren, liegt natürlich in der Verwendung einer gemeinsamen Programmiersprache. Ein häufiger Kandidat für diese Rolle ist Fortran, doch zeigt sich in der Praxis immer wieder, dass Fortran Programme weit weniger transportabel sind als anfänglich angenommen wird. Dies rührt vor allem davon her, dass der Systemprogrammierer gezwungen ist, Erweiterungen zu benutzen, die nur für seine Anlage und deren Fortran Version gelten. Ferner bietet Fortran wenig Schutz vor der Nutzbarmachung bestimmter Eigenschaften der verwendeten Anlage, die durch Fortran hindurchschimmern. Als Folge davon ist die nachträgliche Anpassung eines Programmes an eine andere Anlage oft ein aufwendiges Unterfangen.

Wir wollen daher zeigen, welche Techniken im Fall verwendet werden, wo wiederum der Compiler in seiner eigenen Sprache formuliert ist. Wir unterscheiden zwei Fälle, die grundsätzlich verschiedenes Vorgehen bedingen. Im ersten Fall werden die Arbeiten der Übertragung ausschliesslich auf der

Quellmaschine M1 vorgenommen. Dies dürfte dann üblich sein, wenn M1 einem Liferanten und M2 einem Auftraggeber gehören. Im zweiten Fall wird angenommen, der Empfänger führe die notwendigen Arbeiten selber auf M2 aus. Es ist dies der aufwendigere und unter normalen Verhältnissen kostspieligere Fall.

Angenommen, die Arbeiten seien auf M1 auszuführen, so benötigen wir als Ausgangsprodukte den Compiler in Quellform (S) und in binärer Form (M1). Das Vorgehen ist in Fig. 20 illustriert. Für die Testphase ist ein Interpreter (Emulator) von M2 auf M1 unerlässlich.

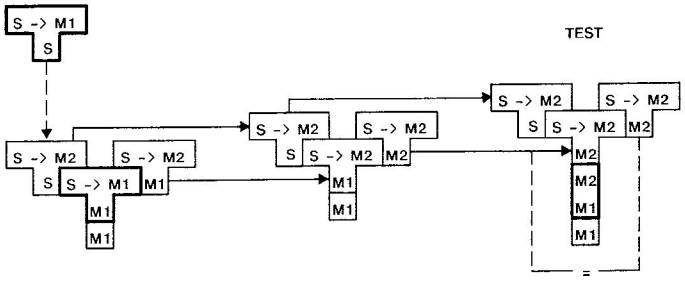


Fig. 20. Transport M1 ' M2 unter Verwendung von M1.

Viel ausgeprägter wird von der Technik der Emulation Gebrauch gemacht, wenn die Arbeiten ausschliesslich auf dem Zielcomputer M2 durchgeführt werden müssen. Hier wird ein Interpreter (Emulator) von M1 auf M2 benötigt (siehe Fig. 21). Der Nachteil dieser Methode ist die ausschliessliche Verwendung interpretativer Compilationen. Immerhin ist zu bedenken, dass sogar ein guter Interpreter für jeden interpretierten Befehl im Durchschnitt zwischen 20 und 30 Maschinenbefehle ausführt. Berücksichtigt man ferner, dass in der Praxis beim Programmieren immer wieder Fehler unterlaufen, und dass daher jeder Schritt in Wirklichkeit etliche Male wiederholt wird, so lassen sich daraus Schlüsse auf den hohen Bedarf an Rechenleistung ziehen. Diese Methode ist daher für die Praxis nicht attraktiv, zumal die zunehmende Verfügbarkeit anderer Computertypen in Service-Zentren mittels Datenfernverbindungen bessere Lösungen offeriert.

Die interpretative Methode besitzt aber doch auch Qualitäten, wenn sie am richtigen Ort eingesetzt wird. So fällt die relativ langsame Ausführung von sehr kurzen Programmen wenig ins Gewicht, weil deren Compilations-, Lade-, und Ein-/Ausgabezeiten die eigentliche Rechenzeit dominieren. Dies hat die

Konsequenz, dass die Wahl eines hypothetischen Computers H und der Bau eines Compilers für S auf H das Problem der Übertragbarkeit wesentlich erleichtern kann. Die einzige Arbeit, die in diesem Fall vom Empfänger aufzuwenden ist, besteht in der Programmierung eines effizienten Interpreters für H auf der Zielmaschine. Der Compiler in Quellform (S) und in compilierter Form (H) bildet zusammen mit einer genauen Beschreibung der hypothetischen Maschine H gleichsam einen Baukasten (compiler kit). Die Übertragung mit Hilfe eines solchen Baukastens stellt eine verhältnismässig einfache Angelegenheit dar.

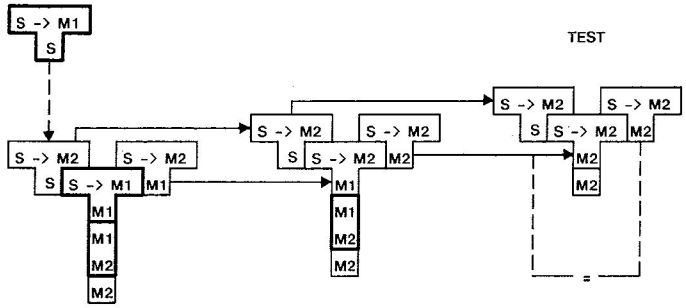


Fig. 21. Transport M1 ' M2 unter Verwendung von M2.

Der untere Teil von Fig. 22 zeigt schliesslich, wie durch Verwendung des Interpreters für H auf M2 wiederum ein echter Compiler erzeugt werden kann. Diesem Vorgehen haftet leider ebenfalls der Nachteil der kostspieligen interpretativen Compilation eines grossen Programms an, nämlich des neuen Compilers selbst.

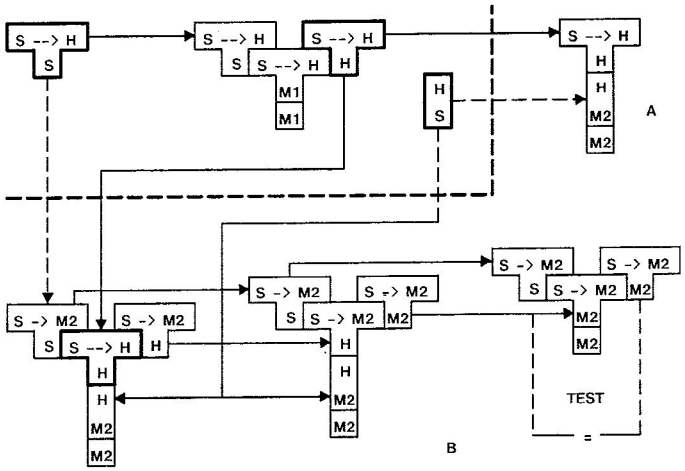


Fig. 22. Transport M1 ← M2 mittels einer hypothetischen Maschine H.

14. Aufgabensammlung

1. Man untersuche die folgende Syntax:

```
S = A.
A = B | "IF" A "THEN" A "ELSE" A.
B = C | B "+" C | "+" C.
C = D | C "&" D | "&" D.
D = "x" | "(" A ")" | "%" D.
```

Bestimme die Mengen der terminalen und nicht-terminalen Symbole, sowie die Mengen der Anfangs- und Folgesymbole für jedes nicht-terminale Symbol. Konstruiere explizit die Folgen von Zerlegungsschritten für die Sätze

```
x + x
(x + x) & (+&x)
(x&%+x)
IF x + x THEN x & x ELSE % x
IF x THEN IF %x THEN x ELSE x + x ELSE x & x
IF % x THEN x ELSE IF x THEN x + x ELSE % x + x
```

2. Erfüllt die Syntax von Aufgabe 1 die Regeln 1 und 2 (s. Kap. 2) für die Zerlegung nach dem top-down Prinzip mit einem lookahead von nur einem Symbol? Wenn nicht, finde eine aequivalente Syntax, die diesen Regeln genügt. Stelle sie durch EBNF Produktionen, als Syntax-Graphen, und als Datenstruktur für Programm 3 dar.

3. Zerlege das PL/0 Programm aus Kapitel 7 explizit in seine syntaktischen Komponenten, und zwar nach dem Prinzip des top-down Parsing.

4. Die folgenden Produktionen definieren einen Ausschnitt aus der Originalversion der Sprache Algol 60. Sie enthält einige Mehrdeutigkeiten, die später in der revidierten Fassung eliminiert wurden. Man ermittle diese Mehrdeutigkeiten anhand einiger Sätze und schlage eine alternative Syntax vor, welche der Sprache eine eindeutige Struktur zuweist.

```
expression = SimpleExpression |
             SimpleExpression relation SimpleExpression |
             "IF" expression "THEN" expression "ELSE" expression .
```

```
relation =      "=" | "#".
SimpleExpression = "a" | "b" | ... | "z".

statement =     BasicStatement | IfStatement | ForStatement.
IfStatement =   "IF" expression "THEN" statement |
                "IF" expression "THEN" statement "ELSE" statement.
ForStatement = "FOR" ForList "DO" statement.
BasicStatement = "A" | "B" | ... | "Z".
ForList =       "10" | "11" | ... | "19".
```

Man untersuche die folgenden Sätze:

IF a THEN b ELSE c = d
IF a THEN IF b THEN A ELSE B
IF a THEN FOR 10 DO IF b THEN A ELSE B

5. Algol 60 enthält eine Mehrfach-Zuweisung von der Form

$v_1 := v_2 := \ldots v_n := e$

Sie sei durch folgende Syntax definiert:

```
assignment =  leftpartlist expression.
leftpartlist =  leftpart | leftpartlist leftpart.
leftpart    =  variable ":=".
expression =  variable | expression "+" variable.
variable    =  ident | ident "[" expression "]".
```

Welches ist der Grad des notwendigen "lookahead", um diese Syntax nach dem top-down Prinzip zu analysieren? Man schlage eine alternative Notation für die Mehrfach-Zuweisung vor, deren lookahead nur 1 Symbol ist.

6. Erstelle ein Programm, welches eine durch EBNF Produktionen dargestellte Syntax einliest und die Mengen der Anfangs-, Folge-, und Endsymbole ermittelt. Ein Endsymbol eines nichtterminalen Symbols A sei ein Symbol, das am Ende eines aus A erzeugten Satzes stehen kann. Hinweis: Man benütze Warshall's Algorithmus (R. W. Floyd, Algorithm 96, Comm. ACM, June 1962).

```
TYPE matrix = ARRAY [1..n],[1..n] OF BOOLEAN;

PROCEDURE ancestor(VAR m: matrix; n: CARDINAL);
(* Initially m[i,j] is TRUE, if individual i is a parent of
```

```
individual j.
  At completion, m[i,j] is TRUE, if i is an ancestor of j *)
   VAR i,j,k: CARDINAL;
BEGIN
  FOR i := 1 TO n DO
    FOR j := 1 TO n DO
      IF m[j,i] THEN
        FOR k := 1 TO n DO
          IF m[i,k] THEN m[j,k] := TRUE END
        END
      END
    END
  END
END ancestor
```

7. Ändere Programm 3 so, dass geprüft wird, ob die eingelesene Syntax den einschränkenden Regeln 1 und 2 genügt. Programm 3 würde z.B. die Syntax

```
A = B C.
B = "x" "y".
C = "x" "z".
```

oder auch

```
A = {"x"} | {"y"}.
```

akzeptieren. Man untersuche zuerst, in welcher Art der Parser in diesen Fällen versagt.

8. Ermittle den PL/0 Code, den der Compiler (Programm 8) für die Prozeduren *divide* und *gcd* (Kap. 7) erzeugt.

9. Man verbessere das PL/0 System von Programm 8, so dass keine Programmierfehler zum Scheitern des Systems führen können. Man beachte insbesondere die Möglichkeiten eines zu kleinen Speichers und eines arithmetischen Überlaufs.

10. Entferne aus der Syntax von PL/0 das Symbol CALL. Dadurch wird die Regel 1 verletzt, und die Syntax muss entsprechend verändert werden. Worauf basiert die Entscheidung, ob eine Anweisung eine Wertzuweisung oder ein Prozeduraufruf ist?

11. Verändere die Sprache PL/0 in eine Variante PL/0', indem die bedingte und die repetierte Anweisung neu formuliert werden:

```
statement =     ident ":=" expression | "CALL" ident |
                "IF" guardedStatements {"|" guardedStatements} "FI" |
                "DO" guardedStatements {"|" guardedStatements} "OD" .
guardedStatements = condition "←" statement {";" statement} .
```

Die neue Anweisung

$$\text{IF } B_0 \leftarrow S_0 \mid B_1 \leftarrow S_1 \mid \ldots \mid B_n \leftarrow S_n \text{ FI}$$

soll besagen, dass von allen Bedingungen B_i, die erfüllt sind, eine beliebige ausgewählt und die zugehörige Anweisung ausgeführt wird. Ist keine erfüllt, so wird das Programm abgebrochen. Jede Anweisungsfolge S_i wird also nur ausgeführt, wenn die Bedingung B_i erfüllt ist; man sagt, S_i sei durch B_i geschützt, und bezeichnet B_i als *guard* von S_i. Die Anweisung

$$\text{DO } B_0 \leftarrow S_0 \mid Bl_1 \leftarrow S_1 \mid \ldots \mid B_n \leftarrow S_n \text{ OD}$$

bedeute, dass irgend ein S_i ausgeführt werde, dessen guard erfüllt ist, und zwar solange, bis kein B_i mehr erfüllt ist. Die DO ... OD Struktur ist also eine repetitive Anweisung.

Die Syntax der Prozedurvereinbarung sollte den neuen Umständen wie folgt angepasst werden.

```
"PROCEDURE" ident ";" statement {";" statement} "END" ";"
```

12. Erweitere Sprache und Compiler PL/0 mit dem Datentyp BOOLEAN und den logischen Operatoren AND, OR, and NOT. Die Syntax ist derart anzupassen, dass anstelle von Bedingungen (conditions) Bool'sche Ausdrücke zugelassen sind. Relationen können als Bool'sche Operanden in Ausdrücken auftreten.

13. Erweitere Sprache und Compiler PL/0 um den Datentyp REAL mit den arithmetischen Operatoren +, -, *, und /. Wähle zwischen den folgenden Alternativen: (a) Das Resultat einer Operation ist stets vom gleichen Typ wie die Operanden. Die Typen INTEGER und REAL lassen sich nicht "mischen". Hingegen gibt es Transferoperatoren zwischen den beiden Typen. (b):

Operanden der Typen INTEGER und REAL lassen sich beliebig mischen. Bei Zuweisungen zu ganzzahligen Variablen wird automatisch gerundet. Vergleiche die Komplexitäten der Implementierung der beiden Fälle.

14. Erweitere die Sprache PL/0 durch Arrays unter der Annahme, dass die Indexgrenzen in der Vereinbarung der Array-Variablen angegeben werden. Die Grenzen seien wie folgt als Konstanten m und n spezifiziert:

VAR a [m : n]

15. Erweitere die Sprache PL/0 durch Prozeduren mit Parametern. Man untersuche die folgenden beiden Möglichkeiten und wähle eine für die Implementation aus.

a. Werteparameter (call by value): Die aktuellen Parameter im Aufruf sind Ausdrücke. Sie werden beim Aufruf ausgewertet. Die formalen Parameter stellen lokale Variablen der Prozedur dar, denen beim Aufruf die Werte der entsprechenden aktuellen Parameter zugewiesen werden.

b. Adressparameter (call by reference): Die aktuellen Parameter sind Variablen, die beim Aufruf für die formalen Parameter substituiert werden. Solche Parameter werden implementiert, indem die Adressen der aktuellen Variablen in Speicherzellen abgelagert werden, die den formellen Parametern zugewiesen sind. Die Variablen werden dann indirekt adressiert. Damit wird es möglich, Variablen ausserhalb der Prozedur mittels der Parameter zu ändern. Es ist damit sinnvoll, die Scope-Regeln der Sprache den neuen Umständen anzupassen: Innerhalb von Prozeduren können nur lokale Grössen direkt zugegriffen werden; nicht-lokale Variablen sind durch Parameter zu erreichen.

16. Verbessere das PL/0 System (Programm 8) durch Einführung eines sogenannten Display. Damit soll der Zugriff zu Variablen beschleunigt werden. Das Display ist ein Array D von Indices, dessen Komponenten D[i] den Ort (Index) des Prozedur-Segmentes auf Niveau i im Stapel angeben. Damit erübrigt sich das Durchlaufen der SL-Kette beim Variablenzugriff, da sich die Adresse aus der einfachen Formel D[i]+a ergibt, wenn i und a Niveau und Offset der Variablen sind. Hingegen muss nun beim Aufruf und am Ende jeder Prozedur das Display auf den neuesten Stand gebracht werden. Kann das Display die SL-Kette ersetzen, oder ist sie dennoch nötig?

17. Führe die in Kapitel 12 behandelte Erweiterung von PL/0 durch Prozesse aus. Um ein brauchbares Werkzeug zu erhalten, ist es unumgänglich, gleichzeitig

auch Prozessparameter und Arrays einzuführen.

18. Ändere das PL/0 System derart, dass anstelle eines Stapels im Interpreter eine konventionelle Computer-Architektur verwendet wird. Dies bedeute insbesondere, dass keine auf den Stapelmechanismus ausgerichtete Befehle mehr vorhanden sind. Es werde vielmehr angenommen, dass mehrere Register zur Verfügung stehen, und dass die Befehle neben einer Speicheradresse noch eine Registernummer enthalten, die das zu verwendende Register angibt.

19. Ändere den Compiler derart, dass Code für einen bestehenden Computer erzeugt wird. Es empfiehlt sich, der Einfachheit halber (mindestens in einem ersten Schritt) symbolischen Assembler-Code zu erzeugen.

20. Erweitere Programm 5 derart, dass es ein beliebiges PL/0 Programm in ein entsprechendes PL/0' Programm übersetzt. Die Syntax von PL/0' ist der Aufgabe 11 zu entnehmen. Ist eine einfache Übersetzung in der umgekehrten Richtung ebenfalls möglich?

Literaturhinweise

Aho, A.V. and Ullman, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, 1977.

Barron, D.W. *Pascal, The Language and its Implementation.* Wiley & Sons, Chichester, 1981.

Brinch Hansen, P. *Operating Systems Principles.* Prentice-Hall, Englewood Cliffs, 1973.

Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

Dijkstra, E.W., Dahl, O.-J., and Hoare, C.A.R. *Structured Programming.* Academic Press, London and New York, 1974.

Earley, J. and Strurgis, H. A formalism for translator interactions. *Comm. ACM 13,* 10 (Oct. 1970), 607-617.

Gries, D. *Compiler Construction for Digital Computers.* McGraw-Hill, New York, 1971.

Kaubisch, W.H., Perrott, R.H. and Hoare, C.A.R. Quasiparallel programming. *Software - Practice and Experience 6,* 3 (July 1976), 341-356.

Knuth, D.E. Top-down syntax analysis. *Acta Informatica 1* (1971), 79-110.

Pemberton, S. and Daniels, M. C. *Pascal Implementation: The P4 Compiler.* Ellis Horwood, Chichester, 1982.

Wirth, N. *Algorithmen und Datenstrukturen.* Teubner-Verlag, Stuttgart, 1983

Wirth, N. *Programming in Modula-2.* Springer-Verlag, New York, 1982.

Wulf, W., et al. *The Design of an Optimizing Compiler.* American Elsevier, New York, 1975.

Zima, H. *Compilerbau.* Bibliographisches Institut, Mannheim, 1982.

Der ASCII Zeichensatz

oct	hex	0 0	20 10	40 20	60 30	100 40	120 50	140 60	160 70
0	0	nul	dle		0	@	P	'	p
1	1	soh	dc1	!	1	A	Q	a	q
2	2	stx	dc2	"	2	B	R	b	r
3	3	etx	dc3	#	3	C	S	c	s
4	4	eot	dc4	$	4	D	T	d	t
5	5	enq	nak	%	5	E	U	e	u
6	6	ack	syn	&	6	F	V	f	v
7	7	bel	etb	'	7	G	W	g	w
10	8	bs	can	(	8	H	X	h	x
11	9	ht	em	)	9	I	Y	i	y
12	A	lf	sub	*	:	J	Z	j	z
13	B	vt	esc	+	;	K	[	k	{
14	C	ff	fs	,	<	L	\	l	|
15	D	cr	gs	-	=	M	]	m	}
16	E	so	rs	.	>	N	↑	n	~
17	F	si	us	/	?	O	←	o	del

Stichwortverzeichnis