ALGORITHMS + DATA STRUCTURES = PROGRAMS

NIKLAUS WIRTH

Eidgenossische Technische Hochschule
Zurich, Switzerland

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, N.J.

Library of Congress Cataloging in Publication Data

WIRTH, NIKLAUS.
Algorithms + data structures = programs.

Bibliography: p.
Includes index.
1. Electronic digital computers—Programming.
2. Data structures (Computer science) 3. Algorithms.
I. Title.
QA76.6.W56 001.6'42 75-11599
ISBN 0-13-022418-9

Current printing (last digit):
19 18 17 16 15 14

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY., LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA (PTE.) LTD., *Singapore*

CONTENTS

PREFACE

In recent years the subject of *computer programming* has been recognized as a discipline whose mastery is fundamental and crucial to the success of many engineering projects and which is amenable to scientific treatment and presentation. It has advanced from a craft to an academic discipline. The initial outstanding contributions toward this development were made by E. W. Dijkstra and C. A. R. Hoare. Dijkstra's "Notes on Structured Programming"* opened a new view of programming as a scientific subject and an intellectual challenge, and it coined the title for a "revolution" in programming. Hoare's "Axiomatic Basis of Computer Programming"† showed in a lucid manner that programs are amenable to an exacting analysis based on mathematical reasoning. Both these papers argue convincingly that many programming errors can be prevented by making programmers aware of the methods and techniques which they hitherto applied intuitively and often unconsciously. These papers focused their attention on the aspects of composition and analysis of programs, or, more explicitly, on the structure of algorithms represented by program texts. Yet, it is abundantly clear that a systematic and scientific approach to program construction primarily has a bearing in the case of large, complex programs which involve complicated sets of data. Hence, a methodology of programming is also bound to include all aspects of data structuring. *Programs*, after all, are concrete formulations of abstract *algorithms* based on particular representations and structures of *data*. An outstanding contribution to bring order into the bewildering variety of terminology and concepts on data structures was made by Hoare through his "Notes on Data Structuring."‡ It made clear that decisions

*In *Structured Programming* by Dahl, Djkstra, and Hoare (New York: Academic Press, 1972), pp. 1–82.

†In *Comm. ACM*, **12**, No. 10 (1969), 576–83.

‡In *Structured Programming*, pp. 83–174

about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often strongly depend on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably intertwined.

Yet, this book starts with a chapter on data structure for two reasons. First, one has an intuitive feeling that data precede algorithms: you must have some objects before you can perform operations on them. Second, and this is the more immediate reason, this book assumes that the reader is familiar with the basic notions of computer programming. Traditionally and sensibly, however, introductory programming courses concentrate on algorithms operating on relatively simple structures of data. Hence, an introductory chapter on data structures seems appropriate.

Throughout the book, and particularly in Chap. 1, we follow the theory and terminology expounded by Hoare* and realized in the programming language PASCAL.† The essence of this theory is that data in the first instance represent abstractions of real phenomena and are preferably formulated as abstract structures not necessarily realized in common programming languages. In the process of program construction the data representation is gradually refined—in step with the refinement of the algorithm—to comply more and more with the constraints imposed by an available programming system.‡ We therefore postulate a number of basic building principles of data structures, called the *fundamental structures.* It is most important that they are constructs that are known to be quite easily implementable on actual computers, for only in this case can they be considered the true elements of an actual data representation, as the *molecules* emerging from the final step of refinements of the data description. They are the *record,* the *array* (with fixed size), and the *set.* Not surprisingly, these basic building principles correspond to mathematical notions which are fundamental as well.

A cornerstone of this theory of data structures is the distinction between fundamental and "advanced" structures. The former are the molecules—themselves built out of atoms—which are the components of the latter. Variables of a fundamental structure change only their value, but never their structure and never the set of values they can assume. As a consequence, the size of the store they occupy remains constant. "Advanced" structures, however, are characterized by their change of value *and* structure during

*"Notes of Data Structuring."

†N. Wirth, "The Programming Language Pascal," *Acta Informatica,* **1**, No. 1 (1971), 35–63.

‡N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM,* **14**, No. 4 (1971), 221–27.

the execution of a program. More sophisticated techniques are therefore needed for their implementation.

The sequential file—or simply the sequence—appears as a hybrid in this classification. It certainly varies its length; but that change in structure is of a trivial nature. Since the sequential file plays a truly fundamental role in practically all computer systems, it is included among the fundamental structures in Chap. 1.

The second chapter treats *sorting algorithms*. It displays a variety of different methods, all serving the same purpose. Mathematical analysis of some of these algorithms shows the advantages and disadvantages of the methods, and it makes the programmer aware of the importance of analysis in the choice of good solutions for a given problem. The partitioning into methods for sorting arrays and methods for sorting files (often called internal and external sorting) exhibits the crucial influence of data representation on the choice of applicable algorithms and on their complexity. The space allocated to sorting would not be so large were it not for the fact that sorting constitutes an ideal vehicle for illustrating so many principles of programming and situations occurring in most other applications. It often seems that one could compose an entire programming course by selecting examples from sorting only.

Another topic that is usually omitted in introductory programming courses but one that plays an important role in the conception of many algorithmic solutions is recursion. Therefore, the third chapter is devoted to *recursive algorithms*. Recursion is shown to be a generalization of repetition (iteration), and as such it is an important and powerful concept in programming. In many programming tutorials it is unfortunately exemplified by cases in which simple iteration would suffice. Instead, Chap. 3 concentrates on several examples of problems in which recursion allows for a most natural formulation of a solution, whereas use of iteration would lead to obscure and cumbersome programs. The class of *backtracking* algorithms emerges as an ideal application of recursion, but the most obvious candidates for the use of recursion are algorithms operating on data whose structure is defined recursively. These cases are treated in the last two chapters, for which the third chapter provides a welcome background.

Chapter 4 deals with *dynamic data structures*, i.e., with data that change their structure during the execution of the program. It is shown that the recursive data structures are an important subclass of the dynamic structures commonly used. Although a recursive definition is both natural and possible in these cases, it is usually not used in practice. Instead, the mechanism used in its implementation is made evident to the programmer by forcing him to use explicit reference or *pointer* variables. This book follows this technique and reflects the present state of the art: Chapter 4 is devoted to

programming with pointers, to lists, trees, and to examples involving even more complicated meshes of data. It presents what is often (and somewhat inappropriately) called "list processing." A fair amount of space is devoted to tree organizations, and in particular to search trees. The chapter ends with a presentation of scatter tables, also called "hash" codes, which are often preferred to search trees. This provides the possibility of comparing two fundamentally different techniques for a frequently encountered application.

The last chapter consists of a concise introduction to the definition of *formal languages* and the problem of *parsing*, and of the construction of a *compiler* for a small and simple language for a simple computer. The motivation to include this chapter is threefold. First, the successful programmer should have at least some insight into the basic problems and techniques of the compilation process of programming languages. Second, the number of applications which require the definition of a simple input or control language for their convenient operation is steadily growing. Third, formal languages define a recursive structure upon sequences of symbols; their processors are therefore excellent examples of the beneficial application of recursive techniques, which are crucial to obtaining a transparent structure in an area where programs tend to become large or even enormous. The choice of the sample language, called PL/0, was a balancing act between a language that is too trivial to be considered a valid example at all and a language whose compiler would clearly exceed the size of programs that can usefully be included in a book that is not directed only to the compiler specialist.

Programming is a constructive art. How can a constructive, inventive activity be taught? One method is to crystallize elementary composition principles out of many cases and exhibit them in a systematic manner. But programming is a field of vast variety often involving complex intellectual activities. The belief that it could ever be condensed into a sort of pure "recipe teaching" is mistaken. What remains in our arsenal of teaching methods is the careful selection and presentation of master examples. Naturally, we should not believe that every person is capable of gaining equally much from the study of examples. It is the characteristic of this approach that much is left to the student, to his diligence and intuition. This is particularly true of the relatively involved and long examples of programs. Their inclusion in this book is not accidental. Longer programs are the "normal" case in practice, and they are much more suitable for exhibiting that elusive but essential ingredient called style and orderly structure. They are also meant to serve as exercises in the art of program *reading*, which too often is neglected in favor of program writing. This is a primary motivation behind the inclusion of larger programs as examples in

their entirety. The reader is led through a gradual development of the program; he is given various "snapshots" in the evolution of a program, whereby this development becomes manifest as a *stepwise refinement* of the details. I consider it essential that programs are shown in final form with sufficient attention to details, for in programming, the devil hides in the details. Although the mere presentation of an algorithm's principle and its mathematical analysis may be stimulating and challenging to the academic mind, it seems dishonest to the engineering practitioner. I have therefore strictly adhered to the rule of presenting the final programs in a language in which they can actually be run on a computer.

Of course, this raises the problem of finding a form which at the same time is both machine executable and sufficiently machine independent to be included in such a text. In this respect, neither widely used languages nor abstract notations proved to be adequate. The language PASCAL provides an appropriate compromise; it had been developed with exactly this aim in mind, and it is therefore used throughout this book. The programs can easily be understood by programmers who are familiar with some other high-level language, such as ALGOL 60 or PL/1, because it is easy to understand the PASCAL notation while proceeding through the text. However, this not to say that some preparation would not be beneficial. The book *Systematic Programming** provides an ideal background because it is also based on the PASCAL notation. The present book was, however, not intended as a manual on the language PASCAL; there exist more appropriate texts for this purpose.†

This book is a condensation—and at the same time an elaboration—of several courses on programming taught at the Federal Institute of Technology (ETH) at Zürich. I owe many ideas and views expressed in this book to discussions with my collaborators at ETH. In particular, I wish to thank Mr. H. Sandmayr for his careful reading of the manuscript, and Miss Heidi Theiler for her care and patience in typing the text. I should also like to mention the stimulating influence provided by meetings of the Working Groups 2.1 and 2.3 of IFIP, and particularly the many memorable arguments I had on these occasions with E. W. Dijkstra and C. A. R. Hoare. Last but not least, ETH generously provided the environment and the computing facilities without which the preparation of this text would have been impossible.

N. WIRTH

*N. Wirth (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1973.)

†K. Jensen and N. Wirth, "PASCAL—User Manual and Report" *Lecture Notes in Computer Science*, Vol. 18 (Berlin, New York; Springer-Verlag, 1974).

5 LANGUAGE STRUCTURES AND COMPILERS

In this chapter we are aiming at developing a compiler (translator) for a simple, rudimentary programming language. This compiler program may serve as an example for the systematic, well-structured development of a program of non-trivial complexity and size. In this respect, it constitutes a welcome application of the program and data structuring disciplines exposed and elaborated in the preceding chapters, but in addition to this, the aim is to present a general introduction to the structure and operation of compilers. Knowledge and insight on this subject will both enhance the general understanding of the art of programming in terms of high-level languages and will make it easier for a programmer to develop his own systems appropriate for specific purposes and areas of application. Since it is well recognized that the discipline of compiler engineering is a complicated and wide subject, the chapter's character in this latter respect will necessarily be introductory and expository. Perhaps the most important single point is that the structure of language is mirrored in the structure of its compiler and that its complexity—or simplicity—intimately determines the complexity of its compiler. We shall therefore start by describing language composition and will then concentrate exclusively on simple structures that lead to simple, modular translators. Language constructs of this kind of structural simplicity are, as it turns out, adequate for virtually all genuine needs arising in practical programming languages.

5.1. LANGUAGE DEFINITION AND STRUCTURE

Every language is based on a *vocabulary*. Its elements are ordinarily called words; in the realm of formal languages, however, they are called

(basic) *symbols*. It is characteristic of languages that some sequences of words are recognized as correct, well-formed *sentences* of the language and that others are said to be incorrect or ill-formed. What is it that determines whether a sequence of words is a correct sentence or not? It is the grammar, syntax, or structure of the language. In fact, we define the *syntax* as the set of rules or formulas which defines the set of (formally correct) sentences. More importantly, however, such a set of rules not only allows us to decide whether or not a given sequence of words is a sentence, but it also provides the sentences with a structure which is instrumental in the recognition of a sentence's meaning. Hence, it is clear that syntax and *semantics* (= meaning) are intimately connected. The structural definitions are therefore always to be considered as auxiliary to a higher purpose. This, however, must not prevent us from initially studying structural aspects exclusively, ignoring the issues of meaning and interpretation.

Take, for example, the sentence, "Cats sleep." The word "cats" is the subject and "sleep" is the predicate. This sentence belongs to the language that may, for instance, be defined by the following syntax.

$$\begin{aligned} &\langle \text{sentence} \rangle &&::= \langle \text{subject} \rangle \langle \text{predicate} \rangle \\ &\langle \text{subject} \rangle &&::= \textit{cats} \mid \textit{dogs} \\ &\langle \text{predicate} \rangle &&::= \textit{sleep} \mid \textit{eat} \end{aligned}$$

The meaning of these three lines is

1. A sentence is formed by a subject followed by a predicate.
2. A subject consists of either the single word "cats" or the word "dogs."
3. A predicate consists of either the word "sleep" or the word "eat."

The idea then is that a sentence may be derived from the *start symbol* ⟨sentence⟩ by repeated application of *replacement rules*.

The formalism or notation in which these rules are written is called *Backus-Naur-Form* (BNF). It was first used in the definition of ALGOL 60 [5-7]. The sentential constructs ⟨sentence⟩, ⟨subject⟩, and ⟨predicate⟩ are called *non-terminal symbols*; the words *cats*, *dogs*, *sleep*, and *eat* are called *terminal symbols*, and the rules are called *productions*. The symbols ::= and | are called *meta-symbols* of the BNF notation. If, for the sake of brevity, we use single capital letters for non-terminal symbols and lower case letters for terminal symbols, then the example can be rewritten as

EXAMPLE 1

$$\begin{aligned} S &::= AB \\ A &::= x \mid y \\ B &::= z \mid w \end{aligned} \tag{5.1}$$

and the language defined by this syntax consists of the four sentences xz, yz, xw, yw.

To be more precise, we present the following mathematical definitions:

1. Let a language $L = L(T, N, P, S)$ be specified by
   (a) A vocabulary T of terminal symbols.
   (b) A set N of non-terminal symbols (grammatical categories).
   (c) A set P of productions (syntactical rules).
   (d) A symbol S (from N), called the start symbol.
2. The language $L(T, N, P, S)$ is the set of sequences of terminal symbols ξ that can be generated from S according to rule 3 below.

$$L = \{\xi \mid S \xrightarrow{*} \xi \quad \text{and} \quad \xi \in T^*\} \tag{5.2}$$

   (we use Greek letters to denote sequences of symbols.) T^* denotes the set of all sequences of symbols from T.
3. A sequence σ_n can be *generated* from a sequence σ_0 if and only if there exist sequences $\sigma_1, \sigma_2, \ldots, \sigma_{n-1}$ such that every σ_i can be directly generated from σ_{i-1} according to rule 4 below:

$$(\sigma_0 \xrightarrow{*} \sigma_n) \longleftrightarrow ((\sigma_{i-1} \longrightarrow \sigma_i) \quad \text{for } i = 1 \ldots n) \tag{5.3}$$

4. A sequence η can be *directly generated* from a sequence ξ if and only if there exist sequences $\alpha, \beta, \xi', \eta'$ such that
   (a) $\xi = \alpha\xi'\beta$
   (b) $\eta = \alpha\eta'\beta$
   (c) P contains the production $\xi' ::= \eta'$

Note: We use $\alpha ::= \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ as a short form for the set of productions $\alpha ::= \beta_1$, $\alpha ::= \beta_2, \ldots,$ $\alpha ::= \beta_n$.

For instance, the sequence xz of Example 1 can be generated by the following sequence of direct generating steps: $S \longrightarrow AB \longrightarrow xB \longrightarrow xz$; hence $S \xrightarrow{*} xz$, and since $xz \in T^*$, xz is a sentence of the language, i.e., $xz \in L$. Note that the non-terminal symbols A and B occur in non-terminating steps only, whereas the terminating step must lead to a sequence that contains *terminal* symbols only. The grammatical rules are called productions because they determine how new forms may be generated or *produced.*

A language is said to be *context free* if and only if it can be defined in terms of a context free production set. A set of productions is context free if and only if all its members have the form

$$A ::= \xi \qquad (A \in N, \quad \xi \in (N \cup T)^*)$$

i.e., if the left side consists of a single non-terminal symbol and can be replaced by ξ regardless of the context in which A occurs. If a production has the form

$$\alpha A \beta ::= \alpha\xi\beta,$$

then it is said to be *context sensitive* because the replacement of A by ξ may

take place only in the context of α and β. We shall subsequently restrict our attention to context free systems.

Example 2 shows how through recursion an infinity of sentences can be generated by a finite set of productions.

EXAMPLE 2

$$\begin{aligned} S &::= xA \\ A &::= z \mid yA \end{aligned} \tag{5.4}$$

The following sentences can be generated from the start symbol S.

$$\begin{array}{l} xz \\ xyz \\ xyyz \\ xyyyz \\ \ldots\ldots \end{array}$$

5.2. SENTENCE ANALYSIS

The task of language translators or processors is primarily not the generation but the *recognition* of sentences and sentence structure. This implies that the generating steps which lead to a sentence must be reconstructed upon reading the sentence, and that its generation steps must be retraced. This is generally a very complicated and sometimes even impossible task. Its complexity intimately depends on the kind of production rules used to define the language. It is the task of the theory of *syntax analysis* to develop recognizing algorithms for languages with rather complicated structural rules. Here, however, our goal is to outline a method for constructing recognizers that are sufficiently simple and efficient to serve in practice. This implies nothing less than that the computational effort to analyze a sentence must be a linear function of the length of the sentence; in the very worst case the dependency function may be $n \cdot \log n$, where n is the sentence length. Clearly, we cannot be bothered with the problem of finding a recognition algorithm for any given language, but we will work pragmatically in the reverse direction: define an efficient algorithm and then determine the class of languages that can be treated by it [5-3].

A first consequence of the basic efficiency requirement is that the choice of every analysis step must depend only on the present state of computation and on a single next symbol being read. Another most important requirement is that no step will have to be revoked later on. These two requirements are commonly known under the technical term *one-symbol-lookahead without backtracking*.

The basic method to be explained here is called *top-down* parsing because it consists of trying to reconstruct the generating steps (which in general form a structural tree) from their start symbol to the final sentence, from the

top down [5-5 and 5-6]. Let us start by revisiting Example 1: We are given the sentence, *Dogs eat*, and we must determine whether or not it belongs to the language. This, by definition, is only the case if it can be generated from the start symbol ⟨sentence⟩. From the grammatical rules it is evident that it can only be a sentence if it is a subject followed by a predicate. We now divide the remaining task; first, we determine whether or not some initial part of the sentence may be generated from the symbol ⟨subject⟩. This is indeed so since *dogs* can be directly generated; the symbol *dogs* is checked off in the input sentence (i.e., we advance our reading position), and we proceed to the second task: checking whether or not the remaining part can be generated from the symbol ⟨predicate⟩. Since this is again the case, the result of the analysis process is affirmative. We may visualize this process by the following trace, showing on the left the tasks still ahead and on the right the part of the input still remaining unread.

⟨sentence⟩	*dogs eat*
⟨subject⟩ ⟨predicate⟩	*dogs eat*
dogs ⟨predicate⟩	*dogs eat*
⟨predicate⟩	*eat*
eat	*eat*
—	—

A second example shows the trace of the analysis process of the sentence *xyyz* according to the productions of Example 2.

S	$xyyz$
xA	$xyyz$
A	yyz
yA	yyz
A	yz
yA	yz
A	z
z	z
—	—

Since the process of retracing the generating steps of a sentence is called *parsing*, what is described above is a *parsing algorithm*. In the two examples the individual replacement steps could decidedly be taken upon inspection of the single next symbol in the input sequence. Unfortunately, this is not always possible, as is evident from the following example:

EXAMPLE 3

$$\begin{aligned} S &::= A \mid B \\ A &::= xA \mid y \\ B &::= xB \mid z \end{aligned} \tag{5.5}$$

We try to parse the sentence *xxxz*

S	$xxxz$
A	$xxxz$
xA	$xxxz$
A	xxz
xA	xxz
A	xz
xA	xz
A	z

and get stuck. The difficulty arises in the very first step in which the decision about replacing S by A or B cannot be taken by looking at the first symbol only. A possible solution lies in simply proceeding according to one of the possible options and to retreat along the taken path of pursued subgoals if no further progress is possible. This action is called *backtracking*. In the language of Example 3, there is no limit to the number of steps that may have to be "undone." This situation is clearly most undesirable; hence, those language structures that lead to backtracking should be identified and avoided in practical applications. Consequently, we shall decree that we will only consider grammar systems satisfying the following restriction that specifies that the initial symbols of alternative right parts of productions be distinct.

RULE 1

Given the production

$$A ::= \xi_1 | \xi_2 | \ldots | \xi_n$$

the sets of initial symbols of all sentences that can be generated from the ξ_i's must be disjoint, i.e.,

$$first(\xi_i) \cap first(\xi_j) = \varnothing \qquad \text{for all } i \neq j.$$

The set $first(\xi)$ is the set of all terminal symbols that can appear in the first position of sentences derived from ξ. Let this set be computed according to the following rules:

1. The first symbol of the argument is terminal:
$$first(a\xi) = \{a\}$$
2. The first symbol is a non-terminal symbol with the derivation rule
$$A ::= \alpha_1 | \alpha_2 | \ldots | \alpha_n$$
Then
$$first(A\xi) = first(\alpha_1) \cup first(\alpha_2) \cup \ldots \cup first(\alpha_n)$$

In Example 3, we notice that $x \in first(A)$ and $x \in first(B)$. Hence Rule 1 is violated by the first production. It is indeed trivial to find a syntax for the

language of Example 3 that satisfies Rule 1. The solution lies in delaying the factoring until all x's have been dealt with. The following productions are *equivalent* with those of (5.5) in the sense that they generate the same set of sentences:

$$\begin{aligned} S &::= C \mid xS \\ C &::= y \mid z \end{aligned} \tag{5.5a}$$

Unfortunately, Rule 1 is not strong enough to shield us from further trouble. Consider

EXAMPLE 4

$$\begin{aligned} S &::= Ax \\ A &::= x \mid \epsilon \end{aligned} \tag{5.6}$$

Here, ϵ denotes the null sequence of symbols. As we try to parse the sentence x, we may proceed into the following "dead alley":

S	x
Ax	x
xx	x
x	—

The trouble arose because we should have followed the production $A ::= \epsilon$ instead of $A ::= x$. This situation is called the *null string problem*, and it arises only for non-terminal symbols that can generate the empty sequence. In order to avoid it, we postulate

RULE 2

For every symbol $A \in N$ which generates the empty sequence ($A \xrightarrow{*} \epsilon$), the set of its initial symbols must be disjoint from the set of symbols that may follow any sequence generated from A, i.e.,

$$first(A) \cap follow(A) = \varnothing$$

The set $follow(A)$ is computed by considering every production P_i of the form

$$X ::= \xi A \eta$$

and taking the set $S_i = first(\eta_i)$. $follow(A)$ is the union of all such sets S_i. If at least one η_i is capable of generating the empty sequence, then the set $follow(X)$ has to be included in $follow(A)$ as well. In Example 4, Rule 2 is violated for the symbol A since

$$first(A) = follow(A) = \{x\}$$

The usual way of expressing a repeated pattern of symbols is by using a recursive definition of a sentential construct. For example, the production

$$A ::= B \mid AB$$

describes the set of sequences $B, BB, BBB, \ldots$. Its use, however, is now

prohibited by Rule 1 because

$$first(B) \cap first(AB) = first(B) \neq \varnothing$$

If we replace the production by the slightly modified version

$$A ::= \epsilon \mid AB$$

generating the sequences ϵ, B, BB, BBB, . . . , we violate Rule 2 because

$$first(A) = first(B)$$

and therefore

$$first(A) \cap follow(A) \neq \varnothing$$

The two restrictive rules obviously prohibit the use of left recursive definitions. A simple method used to avoid these forms is by either using right recursion

$$A ::= \epsilon \mid BA$$

or by extending the symbolism of BNF to allow to express replication explicitly; we shall do so by letting $\{B\}$ denote the set of sequences

$$\epsilon, B, BB, BBB, \ldots$$

Of course, one must be aware that every such construct is capable of generating the empty sequence. (The brackets { and } are meta-symbols of extended BNF.)

From the preceding argument and from the transformation of the productions (5.5) into (5.5a) it may appear that the "trick" of transforming grammars might be the panacea to all problems of syntax analysis. We must, however, keep in mind that sentential structure is instrumental in defining sentential meaning, that explanations of the meaning of a sentential construct are usually expressed in terms of the meaning of the sentential components. Take, for example, the language of expressions consisting of operands a, b, c and the minus sign meaning subtraction.

$$S ::= A \mid S-A$$
$$A ::= a \mid b \mid c$$

According to this grammar, the sentence $a-b-c$ has a structure that can be expressed by using parentheses as follows: $((a-b)-c)$. However, if the grammar is transformed into the syntactically equivalent but left recursion free form

$$S ::= A \mid A-S$$
$$A ::= a \mid b \mid c$$

then the same sentence would be given another structure, namely, the one expressed as $(a-(b-c))$. Considering the conventional meaning of subtraction, we see that the two forms are not at all semantically equivalent.

The lesson, then, is that when defining a language with an inherent meaning, one must always be aware of the *semantic structure* when devising its syntactic structure because the latter must reflect the former.

5.3. CONSTRUCTING A SYNTAX GRAPH

In the previous paragraph, a top-down recognition algorithm was presented that is applicable to grammars satisfying the restrictive Rules 1 and 2. We now turn to the problem of converting this algorithm into a concrete program. There are two essentially different techniques that can be applied. One is to design a general top-down parsing program valid for all possible grammars (satisfying Rules 1 and 2). In this case, particular grammars are to be supplied in the form of some data structure, on the basis of which the program operates. This general parser is in some sense controlled by the data structure; the program is then called *table driven*. The other technique is to develop a top-down parsing program which is specific for the given language and to construct it systematically according to a set of rules which map a given syntax into a sequence of statements, i.e., into a program. Both techniques have their advantages and disadvantages; both will be introduced subsequently. In the development of a compiler for a given programming language the high degree of flexibility and parametrization of the general parser are hardly needed, whereas the specific parser approach usually leads to more efficient and more easily manageable systems and is therefore preferable. In both cases it is advantageous to represent the given syntax by a so-called *recognition-* or *syntaxgraph*. This graph reflects the flow of control during the process of parsing a sentence.

It is a characteristic of the top-down approach that the *goal* of the parsing process is known at the start. The goal is to recognize a sentence, i.e., a sequence of symbols generatable from the start symbol. The application of a production, that is, the replacement of a single symbol by a sequence of symbols, corresponds to the splitting up of a single goal into a number of subgoals to be pursued in specified order. The top-down method is therefore also called *goal-oriented* parsing. In constructing a parser it is easy to take advantage of this obvious correspondence of non-terminal symbols and goals: we construct a subparser for each non-terminal symbol. Each subparser has the goal of recognizing a subsentence generatable from its corresponding non-terminal symbol. Since we wish to construct a graph to represent the total parser, each non-terminal will be mapped into a subgraph. This leads us to the following rules for constructing a recognizer graph.

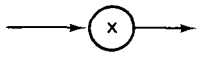
RULES OF GRAPH CONSTRUCTION:

A1. Each nonterminal symbol A with corresponding production set

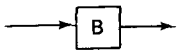
$$A ::= \xi_1 | \xi_2 | \ldots | \xi_n$$

is mapped into a recognition graph A, whose structure is determined by the righthand side of the production according to Rules A2 through A6.

A2. Every occurrence of a *terminal* symbol x in a ξ_i corresponds to a recognizing statement for this symbol and the advancing of the reader to the next symbol of the input sentence. This is represented in the graph by an edge labelled x enclosed in a circle.



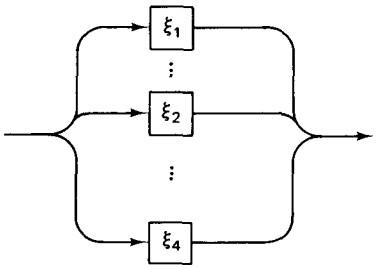
A3. Every occurrence of a *non-terminal* symbol B in a ξ_i corresponds to an activation of the recognizer B. This is represented in the graph by an edge labelled B:



A4. A production having the form

$$A ::= \xi_1 | \ \ldots | \xi_n$$

is mapped into the graph

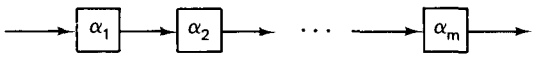


where every $\boxed{\xi_i}$ is obtained by applying construction Rules A2 through A6 to ξ_i.

A5. A ξ having the form

$$\xi = \alpha_1 \alpha_2 \ldots \alpha_m$$

is mapped into the graph

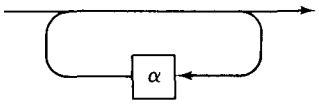


where every $\boxed{\alpha_i}$ is obtained by applying construction Rules A2 through A6 to α_i.

A6. A ξ having the form

$$\xi = \{\alpha\}$$

is mapped into the graph



where $\boxed{\alpha}$ is obtained by applying construction Rules A2 through A6 to α.

EXAMPLE 5

$$\begin{aligned} A &::= x \mid (B) \\ B &::= AC \\ C &::= \{+A\} \end{aligned} \tag{5.7}$$

Here, $+$, x, (, and) are the terminal symbols, whereas { and } belong to the extended BNF and, hence, are meta-symbols. The language generatable from A consists of expressions with operands x, operator $+$, and parentheses. Examples of sentences are

x
(x)
$(x+x)$
$((x))$
. . . . . .

The graphs resulting from the application of the six construction rules are shown in Fig. 5.1. Note that it is possible to reduce this system of graphs into a single graph by suitable substitution of C in B and of B in A (see Fig. 5.2).

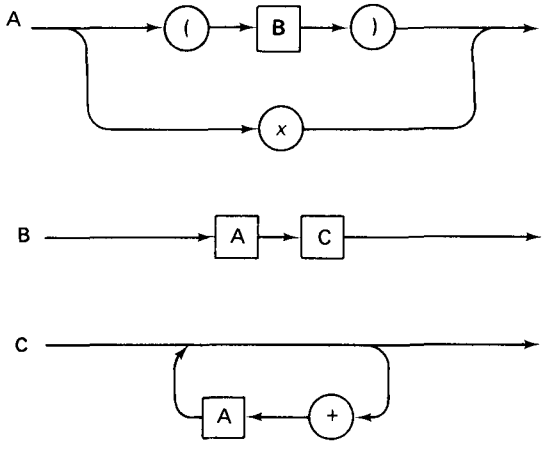


Fig. 5.1 Syntax graphs according to Example 5.

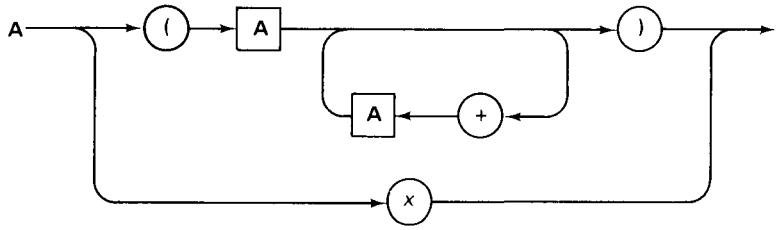


Fig. 5.2 Reduced syntax graph corresponding to Example 5.

The recognition graph is an equivalent representation of the language grammar; it can be used instead of the set of productions in BNF. It is a very convenient form and in many (if not most) instances preferable to BNF. It certainly gives a clearer and more concise picture of a language structure and also conveys a more direct understanding of the parsing process. *The graph is an appropriate form for the designer of a language to start from.* Examples of syntax specifications of entire languages are shown in Sect. 5.7 for PL/0, and in Appendix B for PASCAL.

Restrictive Rules 1 and 2 were imposed in order to allow for deterministic parsing with only one symbol lookahead. How are these rules manifested in the graph representation? It is in this respect that the clarity of the graph becomes most obvious:

1. Rule 1 translates into the requirement that at every fork the branch to be pursued must be selectable by looking only at the next symbol on this branch. This implies that no two branches must start with the same next symbol.
2. Rule 2 translates into the requirement that if any graph A can be traversed without reading an input symbol at all, then this "null branch" must be labelled with all symbols that may follow A. (This will affect the decision to be made upon entering this branch).

It is simple to verify, whether or not a system of graphs satisfies these two adapted rules, without resorting to a BNF representation of the grammar. As an auxiliary step, the sets *first*(A) and *follow*(A) are determined for each graph A. Application of Rules 1 and 2 is then immediate. We call a system of graphs that satisfies these two rules a *deterministic syntax graph.*

5.4. CONSTRUCTING A PARSER FOR A GIVEN SYNTAX

A program which accepts and parses a language is readily derived from its deterministic syntax graph (if such a graph exists). The graph essentially represents the flowchart of the program. In developing this program, how-

ever, one is well-advised to strictly follow a given set of translation rules similar to those that may have led from a BNF to a graph representation of the syntax in the first place. These rules are listed below. They are applicable in a specific framework. This framework consists of a main program in which the procedures corresponding to the various subgoals are embedded and of a routine to proceed to the next symbol.

For the sake of simplicity, let us assume that the sentence to be parsed is represented by the file *input* and that terminal symbols are individual characters. We now postulate the existence within this framework of a character variable *ch* that always represents the next symbol being read. Stepping to the next symbol is then expressed by the statement

read(*ch*)

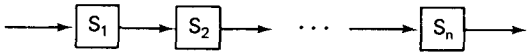
The main program now consists of an initial statement to read the first character, followed by a statement activating the main parsing goal. The individual routines corresponding to parsing goals or graphs are obtained by obeying the following rules. Let the statement obtained by translating the graph S be denoted by $T(S)$.

RULES OF GRAPH TO PROGRAM TRANSLATION:

B1. Reduce the system of graphs to as few individual graphs as possible by appropriate substitutions.

B2. Translate each graph into a procedure declaration according to the subsequent rules B3 through B7.

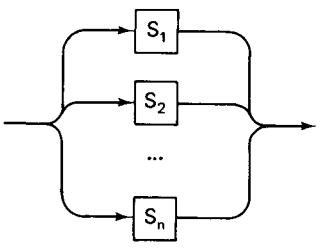
B3. A *sequence* of elements



is translated into the compound statement

begin $T(S_1)$; $T(S_2)$; ...; $T(S_n)$ **end**

B4. A *choice* of elements

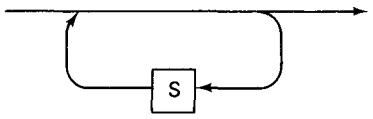


is translated into the selective or conditional statement

case *ch* **of** L_1 : $T(S_1)$; L_2 : $T(S_2)$; . . . . . . L_n : $T(S_n)$ **end**	**if** *ch* **in** L_1 **then** $T(S_1)$ **else** **if** *ch* **in** L_2 **then** $T(S_2)$ **else** . . . . . . **if** *ch* **in** L_n **then** $T(S_n)$ **else** *error*

where L_i denotes the set of initial symbols of the construct $S_i(L_i = first(S_i))$. *Note*: if L_i consists of a single symbol a, then of course "*ch* **in** L_i" should be expressed as "$ch = a$".

B5. A loop of the form

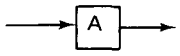


is translated into the statement

while *ch* **in** L **do** $T(S)$

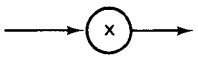
where $T(S)$ is the translation of S according to rules B3 through B7, and L is the set $L = first(S)$ (see preceding note).

B6. An element of the graph denoting another graph A



is translated into the procedure call statement A.

B7. An element of the graph denoting a terminal symbol x



is translated into the statement

if $ch = x$ **then** *read*(*ch*) **else** *error*

where *error* is a routine called when an ill-formed construct is encountered.

The application of these rules is now demonstrated by translating the reduced graph of Example 5 (Fig. 5.2) into a recognizer program (Program 5.1).

```
program parse (input, output);
   var ch: char;
   procedure A;
   begin if ch = 'x' then read(ch) else
            if ch = '(' then
            begin read(ch); A;
               while ch = '+' do
                  begin read(ch); A
                  end ;
               if ch = ')' then read(ch) else error
            end else error
   end ;
begin read(ch); A
end
```

Program 5.1 Parsing Program for Grammar of Example 5.

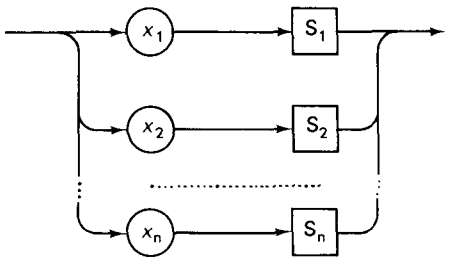
During this translation some obvious programming rules have been freely applied in order to simplify the program. A literal translation would have resulted, for example, in the fourth line reading

```
if ch = 'x' then
      if ch = 'x' then read(ch) else error
else ....
```

which can obviously be reduced into the simpler form presented in the program. Also the read statements in the fifth and seventh lines resulted from similar reductions.

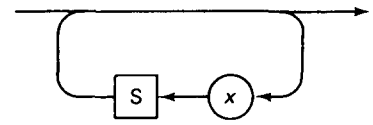
It seems sensible to find out where such reductions are possible in general and then to represent them directly in terms of the graph. The two relevant cases are covered by the following additional rules:

B4a



```
if ch = 'x1' then begin read(ch); T(S1) end else
if ch = 'x2' then begin read(ch); T(S2) end else
   ..........
if ch = 'xn' then begin read(ch); T(Sn) end else error
```

B5a




```
while ch = 'x' do
   begin read(ch); T(S) end
```

In addition, the frequently occurring construct

```
read(ch); T(S);
while B do
   begin read(ch); T(S) end
```

can of course be expressed by the shorter form

repeat *read(ch)*; $T(S)$ **until** B (5.8)

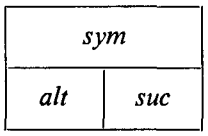
The procedure *error* has so far been left unspecified on purpose. Since we are now only interested in finding out whether an input sequence is well- or ill-formed, we may think of this procedure as a program terminator. Naturally, in practice, more refined principles of coping with ill-formed sentences have to be used. This will be the subject of Sect. 5.9.

5.5. CONSTRUCTING A TABLE-DRIVEN PARSING PROGRAM

Instead of composing a specific program according to the rules given in the preceding chapter for each language and syntax that arises, one may construct a single, general parsing program. Individual language grammars are then fed to the general program in the form of initial data preceding the sentences that are to be parsed. The general program strictly follows the rules of the simple top-down parsing method, and it is straightforward if the underlying syntax graph is deterministic, that is, if the grammar is such that sentences can be parsed with one symbol of lookahead and without backtracking.

Hence, the grammar, which we assume to be represented in the form of a deterministic set of syntax graphs, is translated into an appropriate data structure instead of into a program structure [5-2]. The natural technique

for representing a graph is by introducing a node for each symbol and by connecting these nodes by pointers. Hence, the "table" is not a simple array structure. The rules which guide the translation are given below and are self-evident. The nodes of the data structure are records of two variants: one for terminal and the other for non-terminal symbols. The former are identified by the terminal symbol for which they stand, the latter by a pointer to the data structure representing the corresponding non-terminal symbol. Both variants contain two pointers, one designating the symbol that follows, the *successor*, and the other forming the list of possible *alternatives*. The resulting data type definition is given in (5.9), and within graphs we will depict a node as



At it turns out, we also need an element to represent the empty sequence, the null symbol. We shall denote it by a terminal element, called *empty*.

```
type pointer = ↑node;
   node =
   record suc,alt: pointer;
      case terminal: boolean of                    (5.9)
         true: (tsym: char);
         false: (nsym: hpointer)
   end
```

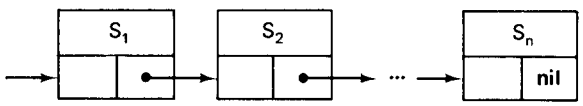
The translation rules from graphs into data structures are analogous to Rules B1 through B7.

RULES OF GRAPH TO DATA STRUCTURE TRANSLATION:

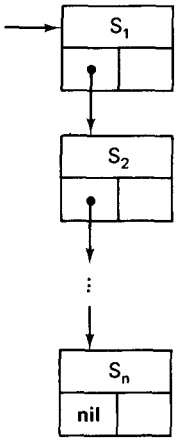
C1. Reduce the system of graphs to as few individual graphs as possible by suitable substitution.

C2. Translate each graph into a data structure according to the subsequent rules C3 through C5.

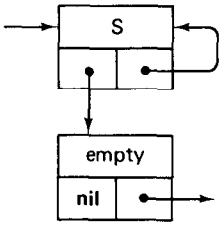
C3. A sequence of elements (see picture of Rule B3) is translated into the following list of data nodes:



C4. The list of alternatives (see picture of Rule B4) is translated into the data structure



C5. A loop (see picture in Rule B5) is translated into the structure



As an example, the graph corresponding to the syntax of Example 5 (Fig. 5.2) results in the structure in Fig. 5.3. The data structure is identified by a *header* node that contains the name of the non-terminal symbol (goal) for which the structure stands. This header is so far unnecessary, for the

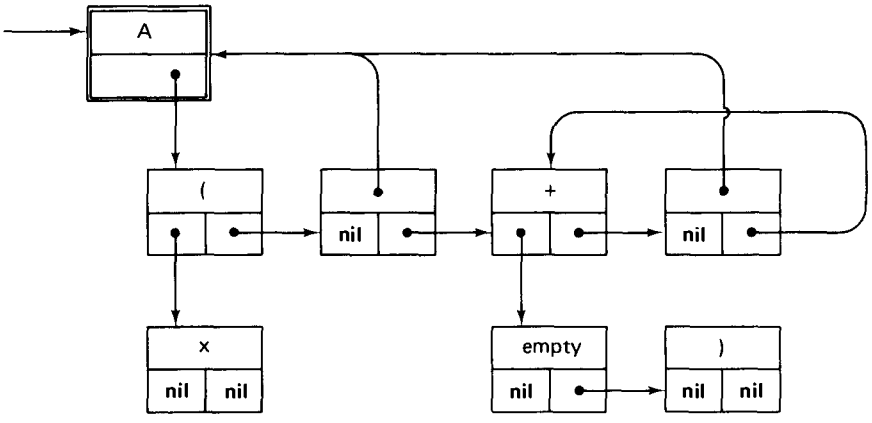


Fig. 5.3 Data structure representing graph of Fig. 5.2.

pointer of the goal field could as well be pointing directly at the "entrance" of the appropriate structure. The header may be used, however, to carry a printable name of the structure.

```
type hpointer = ↑header;
   header =
      record entry: pointer;                    (5.10)
         sym: char
      end
```

The program to parse a sentence—represented as a sequence of characters on the input file—now consists of a repeated statement describing the transition from one node to a next node. The program is expressed as a procedure describing the interpretation of a graph; if a node representing a non-terminal symbol is encountered, then the interpretation of that graph precedes the completion of the interpretation of the present graph. Hence, the interpretation procedure is activated *recursively*. If the current symbol (*sym*) in the input file matches the symbol in the current node of the data structure, then the *suc* field is selected to indicate the next step, otherwise the *alt* field.

```
procedure parse(goal: hpointer; var match: boolean);
   var s : pointer;
begin s := goal↑.entry;
   repeat
      if s↑.terminal then
         begin if s↑.tsym = sym then
               begin match := true; getsym
               end                                  (5.11)
            else match := (s↑.tsym = empty)
         end
      else parse(s↑.nsym, match);
      if match then s := s↑.suc else s := s↑.alt
   until s = nil
end
```

The parsing program (5.11) has the property of immediately pursuing a new subgoal G whenever one appears, without first inspecting whether or not the current symbol is contained in the set of initial symbols *first*(G). This implies that the underlying syntax graph must be void of choices between several alternative non-terminal elements. In particular, if a non-terminal symbol is capable of generating the empty sequence, then none of its right parts must start with a non-terminal symbol.

More sophisticated table-driven parsers may readily be derived from (5.11) which operate on less restrictive classes of grammars. Only slight modifications will also enable it to perform backtracking with, however, notable loss of effectiveness.

Representing a syntax by a graph has one decisive disadvantage: computers cannot directly read graphs. But the data structure that drives the parser must somehow be constructed before parsing can start. It is in this respect that the BNF-representation of grammars appears ideal—as input form for general parsing programs. The next section is therefore devoted to the design of a program which reads a sequence of BNF-productions and transforms them according to rules B1 through B6 into an internal data structure, upon which the parser (5.11) can operate [5-8].

5.6. A TRANSLATOR FROM BNF INTO PARSER-DRIVING DATA STRUCTURES

A translator accepting BNF-productions, converting them into some other representation, is a genuine example of a program whose input data can be regarded as sentences belonging to a language. In fact, it is most appropriate to consider BNF as a language itself, characterized by its own syntax that may, of course, once again be specified in terms of BNF-productions. As a consequence, this translator may serve as a further example of the construction of a recognizer that is, moreover, extended into a translator, or, in general, a processor of its input. Therefore, we shall proceed in the following manner:

Step 1. Define a syntax of the meta language, called EBNF (for Extended BNF).

Step 2. Construct a recognizer for EBNF according to the rules given in Sect. 5.4.

Step 3. Extend the recognizer into a translator, combining it with the table-driven parser.

Let the meta-language—the language of syntax productions—be described by the following productions:

$$
\begin{array}{lll}
\langle \text{production} \rangle & ::= & \langle \text{symbol} \rangle = \langle \text{expression} \rangle . \\
\langle \text{expression} \rangle & ::= & \langle \text{term} \rangle \ \{, \langle \text{term} \rangle\} \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \ \{ \langle \text{factor} \rangle\} \\
\langle \text{factor} \rangle & ::= & \langle \text{symbol} \rangle \ | \ [\langle \text{term} \rangle]
\end{array}
\tag{5.12}
$$

Note that symbols different from the usual BNF meta-symbols have been used to denote exactly these symbols in the production input language. There are two reasons for this:

1. To distingish meta-symbols and language symbols in (5.12).
2. To use characters more commonly available on computing equipment and, in particular, to be able to use the single character = instead of ::=.

The correspondences of usual BNF with our input form are shown in Table 5.1. In addition, each production is terminated by an explicit period.

BNF	Input EBNF
::=	=
|	,
{	[
}	]

Table 5.1 Meta and Language Symbols.

Using this input language to describe the syntax of Example 5 (5.7), we obtain

$$\begin{aligned} A &= x, (B). \\ B &= AC. \\ C &= [+A]. \end{aligned} \tag{5.13}$$

In order to simplify the translator to be constructed, we postulate that terminal symbols be *single letters* and that each production be written on a separate line. This includes the possibility of using blanks in the input (to make it more readable) and of ignoring these blanks by the translator. However, the statement *read(ch)* in Rule B7 must now be replaced by a call to a routine that obtains the next relevant character. This is a very simple case of what is generally called a lexical scanner, or simply a *scanner*. The purpose of a scanner is to extract the next *symbol*—as defined by language representation rules—from the input sequence of *characters*. So far we have considered symbols to be identical to characters; this, however, is a special case and is rarely done in practice.

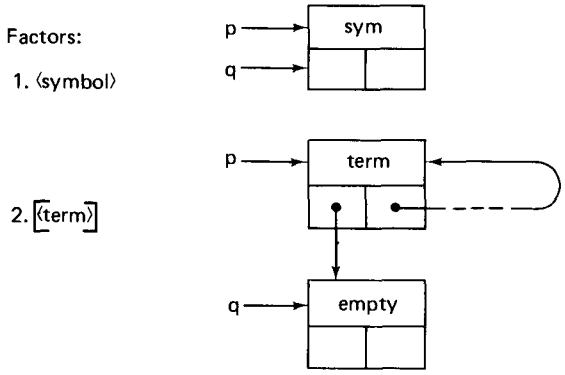
As a last rule, in the input-BNF we postulate that non-terminal symbols be represented by the letters A through H and terminal symbols by the letters I through Z. This is merely a rule of convenience that has no deeper reasons. But it makes it unnecessary to list the vocabularies of terminal and non-terminal symbols prior to the list of productions.

By proceeding strictly according to the parser construction rules B1 through B7, and after having verified that (5.12) satisfies the Restrictive Rules 1 and 2, we obtain Program 5.2 as a recognizer for the language specified by (5.12). Note that the scanner is called *getsym*.

```
program parser(input, output);
label 99;
const empty = '*';
var sym: char;

procedure getsym;
begin
   repeat read(sym); write(sym) until sym ≠ ' '
end {getsym} ;

procedure error;
begin writeln;
   writeln (' INCORRECT INPUT'); goto 99
end {error} ;

procedure term;
   procedure factor;
   begin
      if sym in ['A' . . 'Z', empty] then getsym else
      if sym = '[' then
      begin getsym; term;
         if sym = ']' then getsym else error
      end else error
   end {factor} ;
begin factor;
   while sym in ['A' . . 'Z', '[', empty] do factor
end {term} ;

procedure expression;
begin term;
   while sym = ',' do
   begin getsym; term
   end
end {expression} ;

begin {main program}
   while ¬eof(input) do
   begin getsym;
      if sym in ['A' . . 'Z'] then getsym else error;
      if sym = '=' then getsym else error;
      expression;
      if sym ≠ '.' then error;
      writeln; readln;
   end ;
99: end .
```

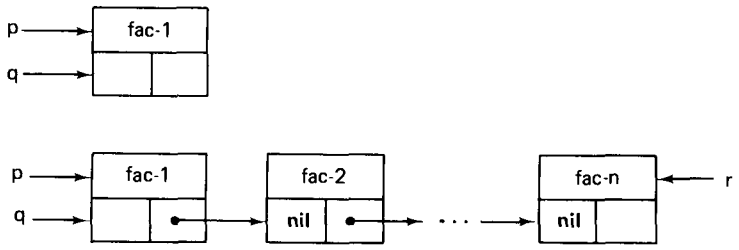
Program 5.2 Parser of Language (5.12).

Step 3 in the development of the translator is concerned with constructing the desired data structure that represents the BNF productions read and that can be interpreted by the parsing procedure (5.11). Unfortunately, this step is not susceptible to formalization as was the step concerned with the recognizer construction. Lacking a formal approach, we describe once again the structures that are desired to represent each language construct by a picture. The resulting structures are then passed as result parameters of the corresponding recognition procedures, which by then have been augmented into translation procedures. It is natural to return as results not the data structures themselves, but pointers p, q, r referring to the structures instead.



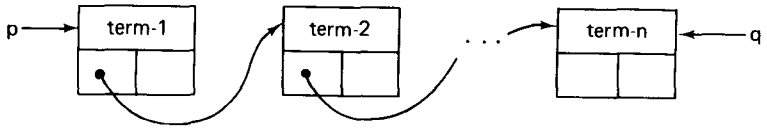
Terms:

⟨factor-1⟩ . . . ⟨factor-n⟩



Expressions:

⟨term-1⟩ ⟨term-2⟩ . . . ⟨term-n⟩



Clearly, it is the task of the procedure *factor* to generate new elements of the data structure; the task of the remaining two procedures is to link them together into a linear list in which *term* uses the *suc* field and *expression* uses the *alt* field for chaining. The details are evident from Program 5.3.

The technique of processing non-terminal symbols needs further clarification. It is possible for a non-terminal symbol to appear as a factor before it appears as a left part in a production. A procedure *find*(*sym*, *h*) is used to locate the symbol *sym* in a linear list in which all headers representing the non-terminal symbols are collected. If a symbol is located, its reference is assigned to *h*; if it is not yet present in the list, it is added to the list. Procedure *find* uses the sentinel technique discussed in detail in Chap. 4.

Program 5.3 consists of three parts, each corresponding to a section of input. Part 1 is concerned with the processing of *productions* into corresponding data structures. Part 2 reads and identifies a single symbol, namely, the one specified as the *start symbol* which generates sentences of the language. (It is preceded by a $ sign delimiting Parts 1 and 2 of the input data.) Part 3 is the parsing program (5.11) reading *input sentences* under control of the data structure generated in Part 1.

It is noteworthy that Program 5.3 has been developed by merely inserting further statements into the *unchanged* Program 5.2. The existing program deals exclusively with the recognition of correctly formed sentences, and it can be used as a framework for the extended program that not only recognizes but also processes or translates the accepted sentences. This method of constructing language processors by *stepwise refinement*, or rather stepwise *enrichment*, is highly recommended. It permits the designer to deal exclusively with a selected aspect of language processing before taking into account other aspects, and hence it facilitates the task of verifying the correctness of a translator program, or at least of maintaining a high confidence level throughout the program's development. In this rather simple example this development consists of only two steps. More complicated languages and more complicated translation tasks require a considerably higher number of individual enrichment steps. A highly similar development in three steps will be the subject of Sects. 5.8 through 5.11.

As evidenced by the development of Program 5.3, the *syntax table-driven* —or rather data structure-driven—approach to parsing provides a degree of freedom and flexibility not presented in the scheme of the specific parser program. This additional flexibility, although not required in general, is the very essence of compilers for so-called *extensible languages*. An extensible language can be extended by further syntactic constructs, more or less at the discretion of the programmer. Analogous to the input of Program 5.3, the input of an extensible language compiler consists of a section specifying the language extensions used in the subsequent program. A more ambitious

scheme even allows altering the language during the process of compilation by intermixing parts of the program to be translated with sections of new language specifications.

As appealing or exciting as these ideas may seem, efforts to realize such compilers have been marked by a notable lack of success. The reason is that the aspect of syntax and of sentence recognition is but a part of the whole task of translation, and in fact even the minor part. It is also the part that is most easily formalized and is therefore most readily represented by a systematized table structure. The much harder part to formalize is the *meaning* of the language, that is, the output or result of translation. This problem has so far not been solved even nearly satisfactorily, which explains why compiler designers tend to be much more enthusiastic about extensible languages before than after their first completed assignment. We conclude our lesson by devoting the remainder of this chapter to developing a modest compiler for one specific, small programming language.

Program 5.3 Translator of Language (5.12).

```
program generalparser (input, output);
label 99;
const empty = '*';
type pointer = ↑node;
     hpointer = ↑header;
   node = record suc, alt: pointer;
                 case terminal: boolean of
                    true: (tsym: char);
                    false: (nsym: hpointer)
          end ;
   header = record sym: char;
                   entry: pointer;
                   suc: hpointer
            end ;
var list, sentinel, h: hpointer;
   p: pointer;
   sym: char;
   ok: boolean;

procedure getsym;
begin
   repeat read(sym); write(sym) until sym ≠ ' '
end {getsym} ;
```

```
procedure find(s: char; var h: hpointer);
{locate nonterminal symbol s in list. if not present, insert it}
   var h1: hpointer;
begin h1 := list; sentinel↑.sym := s;
   while h1↑.sym ≠ s do h1 := h1↑.suc;
   if h1 = sentinel then
      begin {insert} new (sentinel);
         h1↑.suc := sentinel; h1↑.entry := nil
      end ;
   h := h1
end {find} ;

procedure error;
begin writeln;
   writeln ('INCORRECT SYNTAX'); goto 99
end {error} ;

procedure term (var p,q,r: pointer);
   var a,b,c: pointer;
   procedure factor (var p,q: pointer);
      var a,b: pointer; h: hpointer;
   begin if sym in ['A' . . 'Z', empty] then
      begin {symbol}  new(a);
         if sym in ['A' . . 'H'] then
         begin {nonterminal}  find(sym,h);
            a↑.terminal := false; a↑.nsym := h
         end else
         begin {terminal}
            a↑.terminal := true; a↑.tsym := sym
         end ;
         p := a; q := a; getsym
      end else
      if sym = '[' then
      begin getsym; term(p,a,b); b↑.suc := p;
         new(b); b↑.terminal := true; b↑.tsym := empty;
         a↑.alt := b; q := b;
         if sym = ']' then getsym else error
      end else error
   end {factor} ;
begin factor(p,a); q := a;
   while sym in ['A' . . 'Z', '[', empty] do
   begin factor(a↑.suc, b); b↑.alt := nil ; a := b
   end ;
   r := a
end {term} ;
```

Program 5.3 (Continued)

```
procedure expression (var p,q: pointer);
   var a,b,c: pointer;
begin term(p,a,c): c↑.suc := nil;
   while sym = ',' do
   begin getsym;
      term(a↑.alt, b, c): c↑.suc := nil; a := b
   end ;
   q := a
end {expression} ;

procedure parse (goal: hpointer; var match: boolean);
   var s: pointer;
begin s := goal↑.entry;
   repeat
      if s↑.terminal then
      begin if s↑.tsym = sym then
               begin match := true; getsym
               end
            else match := (s↑.tsym = empty)
      end
      else parse(s↑.nsym, match); .
      if match then s := s↑.suc else s := s↑.alt
   until s = nil
end {parse} ;

begin {productions}
   getsym; new(sentinel); list := sentinel;
   while sym ≠ '$' do
   begin find(sym,h);
      getsym; if sym = '=' then getsym else error;
      expression (h↑.entry, p); p↑.alt := nil;
      if sym ≠ '.' then error;
      writeln; readln; getsym
   end ;
   h := list; ok := true;  {check whether all symbols are defined}
   while h ≠ sentinel do
   begin if h↑.entry = nil then
         begin writeln(' UNDEFINED SYMBOL ', h↑.sym);
               ok := false
         end ;
      h := h↑.suc
   end ;
```

Program 5.3 (Continued)

```
    if ¬ok then goto 99;
{goal symbol}
    getsym; find(sym,h); readln; writeln;
{sentences}
    while ¬eof(input) do
        begin write(' '); getsym; parse(h,ok);
            if ok ∧ (sym='.') then writeln (' CORRECT')
                              else writeln (' INCORRECT');
            readln
        end ;
99: end .
```

Program 5.3 (Continued)

5.7. THE PROGRAMMING LANGUAGE PL/0

The remaining sections of this chapter are devoted to the development of a compiler for a language to be called PL/0. The necessity of keeping this compiler reasonably small in order to fit into the framework of this book and the desire to be able to expose the most fundamental concepts of compiling high-level languages constitute the boundary conditions for the design of this language. There is no doubt that either an even simpler or a much more complicated language could have been chosen; PL/0 is one possible compromise between sufficient simplicity to make the exposition transparent and sufficient complexity to make the project worthwhile. A considerably more complicated language is PASCAL, whose compiler was developed using the same techniques, and whose syntax is shown in Appendix B.

As far as program structures are concerned, PL/0 is relatively complete. It features, of course, the assignment statement as the basic construct on the statement level. The structuring concepts are those of sequencing, conditional execution and repetition, represented by the familiar forms of **begin/end-**, **if-**, and **while** statements. PL/0 also features the subroutine concept and, hence, contains a procedure declaration and a procedure call statement.

In the realm of data types, however, PL/0 adheres to the demand for simplicity without compromise: integers are its only data type. It is possible to declare constants and variables of this type. Of course, PL/0 features the conventional arithmetic and relational operators.

The presence of procedures, that is, of more or less "self-contained" partitions of a program offers the opportunity to introduce the concept of *locality* of objects (constants, variables, and procedures). PL/0 therefore

features declarations in the heading of each procedure, implying that these objects are understood to be local to the procedure in which they are declared.

This brief introduction and overview provide the necessary intuition to understand the syntax of PL/0. This syntax is presented in Fig. 5.4 in the form of seven diagrams. The task of transforming the diagrams into a set of equivalent BNF-productions is left to the interested reader. Fig. 5.4 is a convincing example of the expressive power of these diagrams which allow formulation of the syntax of an entire programming language in such a concise and readable form.

The following PL/0 program may demonstrate the use of some features that are included in this mini-language. The program contains the familiar algorithms for multiplication, division, and finding the greatest common divisor (gcd) of two natural numbers.

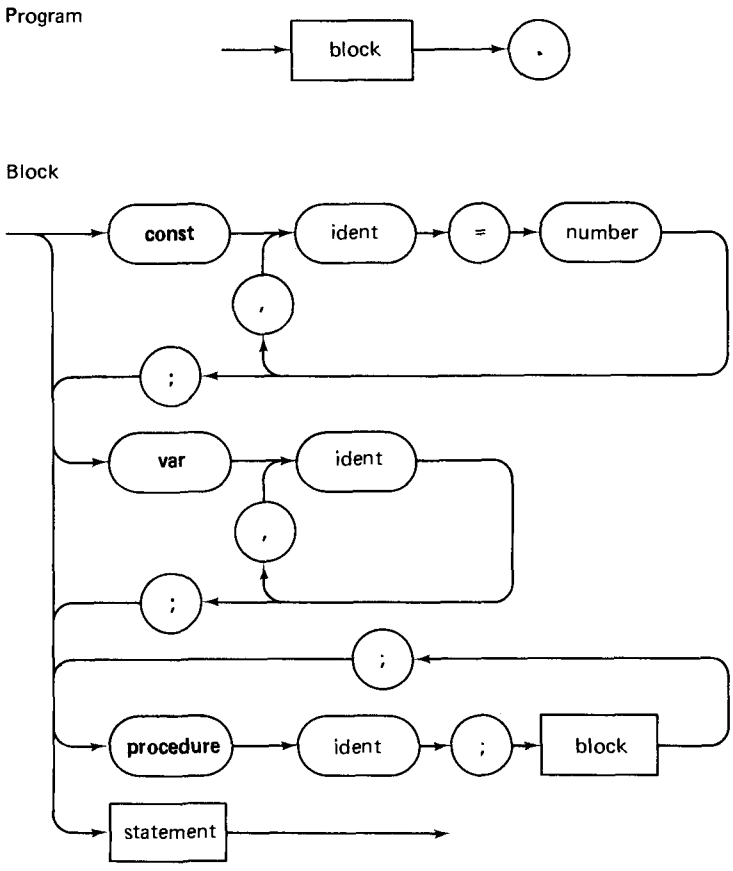
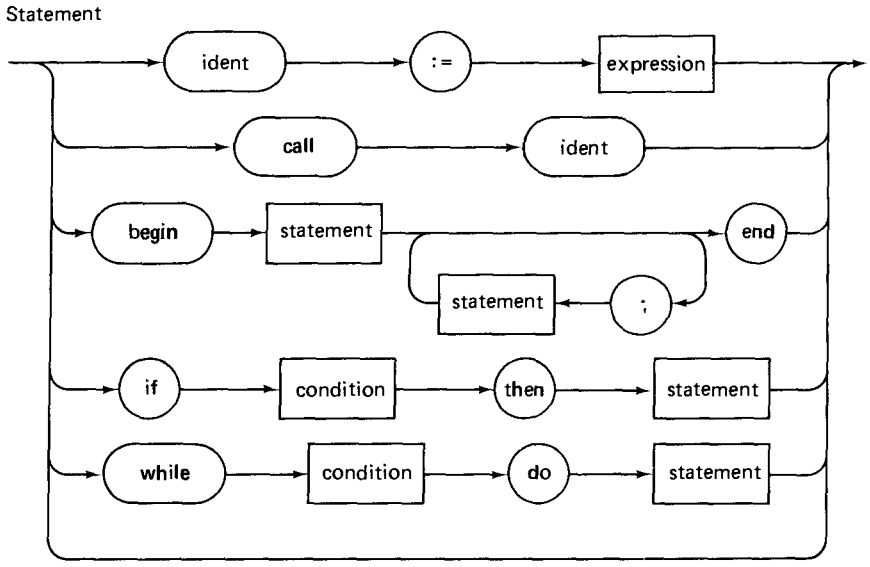


Fig. 5.4 Syntax of PL/0.



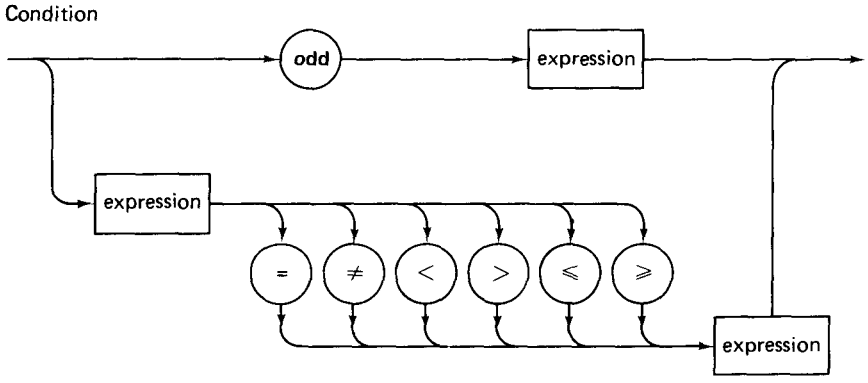


Fig. 5.4 (Continued)

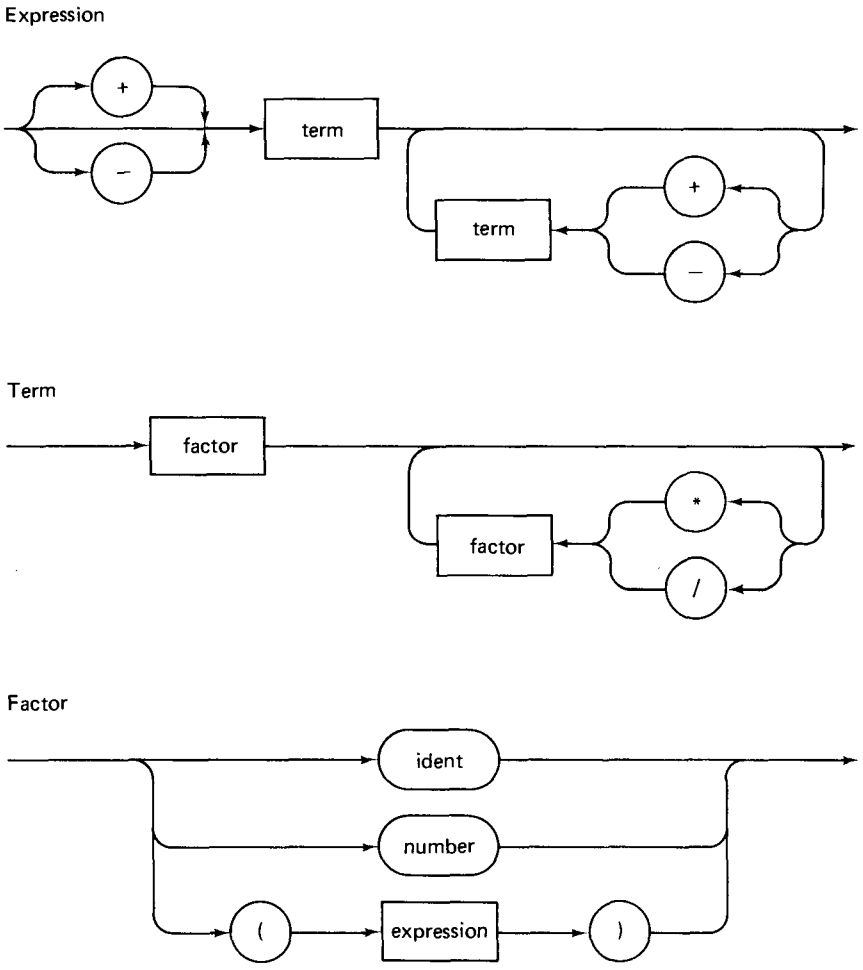


Fig. 5.4 (Continued)

```
const m = 7, n = 85;
var x,y,z,q,r;
procedure multiply;
    var a,b;
begin a := x; b := y; z := 0;
        while b > 0 do
        begin                                          (5.14)
            if odd b 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;                    (5.15)
            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;                (5.16)
              if g < f then f := f−g;
        end ;
    z := f
end ;

begin
    x := m; y := n; call multiply;
    x := 25; y := 3; call divide;
    x := 84; y := 36; call gcd;
end .
```

5.8. A PARSER FOR PL/0

As a first step toward the PL/0 compiler a parser is being developed. This can be done strictly according to the parser Construction Rules B1 through B7 outlined in Sect. 5.4. This method, however, is only applicable if the Restrictive Rules 1 and 2 are satisfied by the underlying syntax. We are therefore obliged to verify this condition, as formulated for their application to syntax graphs.

Rule 1 specifies that every branch emanating from a fork point must lead toward a distinct first symbol. This is very simple to verify on the syntax diagrams of Fig. 5.4. Rule 2 applies to all graphs that can be traversed without reading any symbol. The only such graph in the PL/0 syntax is the one describing statements. Rule 2 demands that all first symbols that may follow a statement must be disjoint from initial symbols of statements. Since later on it will be useful to know the sets of initial and following symbols for all graphs, we shall determine these sets for all seven non-terminal symbols (graphs) of the PL/0 syntax (except for "program"). Table 5.2 provides the

Non-terminal Symbol S	Initial Symbols $L(S)$	Follow Symbols $F(S)$
Block	**const var** **procedure** *ident* **if call begin while**	. ;
Statement	*ident* **call** **begin if while**	. ; **end**
Condition	**odd** + − (*ident number*	**then do**
Expression	+ − (*ident number*	. ;) R **end then do**
Term	*ident number* (	. ;) R + − **end then do**
Factor	*ident number* (	. ;) R + − * / **end then do**

Table 5.2 Initial and Follow Symbols in PL/0.

desired assurance, namely, that the sets of initial and following symbols of statements do not intersect. Application of the parser Construction Rules B1 through B7 is thereby legalized.

The careful reader will have noticed that the basic symbols of PL/0 are no longer single characters as in the preceding examples. Instead, the basic symbols are themselves sequences of characters, such as BEGIN, or :=. As in Program 5.3, a so-called scanner is used to take care of the merely representational or lexical aspects of the input sequence of symbols. The scanner is conceived as a procedure *getsym* whose task is to get the next symbol. The scanner serves the following purposes:

1. It skips separators (blanks).
2. It recognizes reserved words, such as BEGIN, END, etc.
3. It recognizes non-reserved words as identifiers. The actual identifier is assigned to a global variable called *id*.
4. It recognizes sequences of digits as numbers. The actual value is assigned to a global variable *num*.
5. It recognizes pairs of special characters, such as :=.

In order to scan the input sequence of characters, *getsym* uses a local procedure *getch* whose task is to get the next character. Apart from this main purpose, *getch* also

1. Recognizes and suppresses line end information.
2. Copies the input onto the output file, thus generating a program listing.
3. Prints a line number or location counter at the beginning of each line.

The scanner constitutes the necessary one-symbol lookahead. Moreover, the auxiliary procedure *getch* represents an additional lookahead of one

character. Therefore, the total lookahead of this compiler is one symbol plus one character.

The details of these routines are evident from Program 5.4 which represents the complete parser for PL/0. In fact, this parser is already extended in the sense that it collects the declared identifiers denoting constants, variables, and procedures in a *table*. The occurrence of an identifier within a statement then causes a search of this table to determine whether or not the identifier had been properly declared. The lack of such a declaration may duly be regarded as a syntactic error since it is a formal error in the composition of the program text because of the use of an "illegal" symbol. The fact that this error can only be detected by retaining information in a table is a consequence of the inherent *context dependence* of the language, manifest in the rule that all identifiers have to be declared in the appropriate context. Indeed, practically all programming languages are context sensitive in this sense; nevertheless, the context-free syntax is a most helpful model for these languages and greatly aids in the systematic construction of their recognizers. The framework thus obtained can then very easily be extended to take care of the few context sensitive elements of the language, as witnessed by the introduction of the identifier table in the present parser.

Before constructing the individual parser procedures corresponding to the individual syntax graphs, it is useful to determine how these graphs depend on each other. To this end, a so-called *dependence diagram* is constructed; it displays the relationships of the individual graphs, i.e., it lists for each graph G all those graphs $G_1 \ldots G_n$ in terms of which G is defined. Correspondingly, it shows those procedures that will be called by other procedures. The dependence graph for PL/0 is shown in Fig. 5.5.

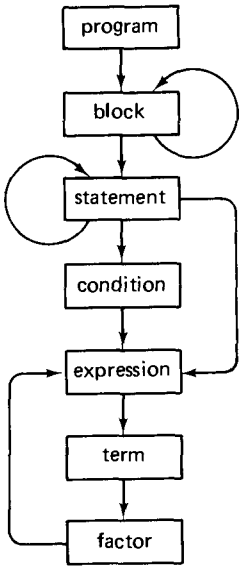


Fig. 5.5 Dependence diagram for PL/0.

The loops in Fig. 5.5 indicate instances of recursion. It is therefore essential that a language in which the PL/0 compiler is implemented is not burdened by prohibition of recursion. In addition, the dependence diagram also allows drawing conclusions on the hierarchical organization of the parser program. For instance, all routines may be contained in (be declared local to) the routine that parses the construct ⟨program⟩ (which is therefore the main program part of the parser). Furthermore, all routines below ⟨block⟩ may be defined locally to the routine representing the parsing goal ⟨block⟩. Naturally, all of these routines call upon the scanner *getsym*, which in turn calls upon *getch*.

Program 5.4 PL/0 Parser.

```
program PL0 (input, output);
{PL/0 compiler, syntax analysis only}
label 99;
const norw = 11;        {no. of reserved words}
   txmax = 100;         {length of indentifier table}
   nmax = 14;           {max. no of digits in numbers}
   al = 10;             {length of identifiers}
type symbol =
   (nul, ident, number, plus, minus, times, slash, oddsym,
    eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
    period, becomes, beginsym, endsym, ifsym, thensym,
    whilesym, dosym, callsym, constsym, varsym, procsym);
   alfa = packed array [1 .. al] of char;
   object = (constant, variable, procedure);
var ch: char;           {last character read}
   sym: symbol;         {last symbol read}
   id: alfa;            {last identifier read}
   num: integer;        {last number read}
   cc: integer;         {character count}
   ll: integer;         {line length}
   kk: integer;
   line: array [1 .. 81] of char;
   a: alfa;
   word: array [1 .. norw] of alfa;
   wsym: array [1 .. norw] of symbol;
   ssym: array [char] of symbol;
   table: array [0 .. txmax] of
          record name: alfa;
                 kind: object
          end ;
```

```
procedure error (n: integer);
begin writeln (' ':cc, '↑', n:2); goto 99
end {error} ;
procedure getsym;
    var i,j,k: integer;
    procedure getch;
    begin if cc = ll then
        begin if eof(input) then
                    begin write (' PROGRAM INCOMPLETE'); goto 99
                    end ;
            ll := 0; cc := 0; write(' ');
            while ¬eoln(input) do
                begin ll := ll+1; read(ch); write(ch); line[ll] := ch
                end ;
            writeln; ll := ll+1; read(line[ll])
        end ;
        cc := cc+1; ch := line[cc]
    end {getch} ;
begin {getsym}
while ch = ' ' do getch;
    if ch in ['A'..'Z'] then
    begin {identifier or reserved word}  k := 0;
        repeat if k < al then
            begin k := k+1; a[k] := ch
            end ;
            getch
        until ¬(ch in ['A'..'Z', '0'..'9']);
        if k ≥ kk then kk := k else
            repeat a[kk] := ' '; kk := kk-1
            until kk = k;
        id := a; i := 1; j := norw;
        repeat k := (i+j) div 2;
            if id ≤ word[k] then j := k-1;
            if id ≥ word[k] then i := k+1
        until i > j;
        if i-1 > j then sym := wsym[k] else sym := ident
    end else
    if ch in ['0'..'9'] then
    begin {number} k := 0; num := 0; sym := number;
        repeat num := 10*num + (ord(ch)-ord('0'));
            k := k+1; getch
        until ¬(ch in ['0'..'9']);
        if k > nmax then error (30)
    end else
```

Program 5.4 (Continued)

```
    if ch = ':' then
    begin getch;
        if ch = '=' then
        begin sym := becomes; getch
        end else sym := nul;
    end else
    begin sym := ssym[ch]; getch
    end
end {getsym} ;
procedure block (tx: integer);
    procedure enter (k: object);
    begin {enter object into table}
        tx := tx + 1;
        with table[tx] do
        begin name := id; kind := k;
        end
    end {enter} ;
    function position (id: alfa): integer;
        var i: integer;
    begin {find identifier id in table}
        table[0].name := id; i := tx;
        while table[i].name ≠ id do i := i-1;
        position := i
    end {position} ;
    procedure constdeclaration;
    begin if sym = ident then
        begin getsym;
            if sym = eql then
            begin getsym;
                if sym = number then
                    begin enter (constant); getsym
                    end
                else error (2)
            end else error (3)
        end else error (4)
    end {constdeclaration} ;
    procedure vardeclaration;
    begin if sym = ident then
            begin enter (variable); getsym
            end else error (4)
    end {vardeclaration} ;
```

Program 5.4 (Continued)

```
procedure statement;
   var i: integer;
   procedure expression;
      procedure term;
         procedure factor;
            var i: integer;
         begin
            if sym = ident then
            begin i := position(id);
               if i = 0 then error (11) else
               if table[i].kind = procedure then error (21);
               getsym
            end else
            if sym = number then
            begin getsym
            end else
            if sym = lparen then
            begin getsym; expression;
               if sym = rparen then getsym else error (22)
            end
            else error (23)
         end {factor} ;
         begin {term} factor;
            while sym in [times, slash] do
               begin getsym; factor
               end
         end {term} ;
      begin {expression}
         if sym in [plus, minus] then
            begin getsym; term
            end else term;
         while sym in [plus, minus] do
            begin getsym; term
            end
      end {expression} ;
      procedure condition;
      begin
         if sym = oddsym then
         begin getsym; expression
         end else
         begin expression;
            if ¬(sym in [eql, neq, lss, leq, gtr, geq]) then
               error (20) else
            begin getsym; expression
            end
         end
      end {condition} ;
```

```
        begin {statement}
        if sym = ident then
        begin i := position(id);
            if i = 0 then error (11) else
            if table [i] .kind ≠ variable then error (12);
            getsym; if sym = becomes then getsym else error (13);
            expression
        end else
        if sym = callsym then
        begin getsym;
            if sym ≠ ident then error (14) else
                begin i := position(id);
                    if i = 0 then error (11) else
                    if table[i] .kind ≠ procedure then error (15);
                    getsym
                end
        end else
        if sym = ifsym then
        begin getsym; condition;
            if sym = thensym then getsym else error (16);
            statement;
        end else
        if sym = beginsym then
        begin getsym; statement;
            while sym = semicolon do
                begin getsym; statement
                end ;
            if sym = endsym then getsym else error (17)
        end else
        if sym = whilesym then
        begin getsym; condition;
            if sym = dosym then getsym else error (18);
            statement
        end
    end {statement} ;
begin {block}
    if sym = constsym then
    begin getsym; constdeclaration;
        while sym = comma do
            begin getsym; constdeclaration
            end ;
        if sym = semicolon then getsym else error (5)
    end ;
```

Program 5.4 (Continued)

```
    if sym = varsym then
    begin getsym; vardeclaration;
        while sym = comma do
            begin getsym; vardeclaration
            end ;
        if sym = semicolon then getsym else error (5)
    end ;
    while sym = procsym do
    begin getsym;
        if sym = ident then
            begin enter (procedure); getsym
            end
        else error (4);
        if sym = semicolon then getsym else error (5);
        block (tx);
        if sym = semicolon then getsym else error (5);
    end ;
    statement
end {block} ;
begin {main program}
    for ch := 'A' to ';' do ssym[ch] := nul;
    word[ 1] := 'BEGIN ';    word[ 2] := 'CALL      ';
    word[ 3] := 'CONST';     word[ 4] := 'DO        ';
    word[ 5] := 'END   ';    word[ 6] := 'IF        ';
    word[ 7] := 'ODD   ';    word[ 8] := 'PROCEDURE';
    word[ 9] := 'THEN  ';    word[10] := 'VAR       ';
    word[11] := 'WHILE ';
    wsym[ 1] := beginsym;    wsym[ 2] := callsym;
    wsym[ 3] := constsym;    wsym[ 4] := dosym;
    wsym[ 5] := endsym;      wsym[ 6] := ifsym;
    wsym[ 7] := oddsym;      wsym[ 8] := procsym;
    wsym[ 9] := thensym;     wsym[10] := varsym;
    wsym[11] := whilesym;
    ssym['+'] := plus;       ssym['-'] := minus;
    ssym['*'] := times;      ssym['/'] := slash;
    ssym['('] := lparen;     ssym[')'] := rparen;
    ssym['='] := eql;        ssym[','] := comma;
    ssym['.'] := period;     ssym['≠'] := neq;
    ssym['<'] := lss;        ssym['>'] := gtr;
    ssym['≤'] := leq;        ssym['≥'] := geq;
    ssym[';'] := semicolon;
    page(output);
    cc := 0; ll := 0; ch := ' '; kk := al; getsym;
    block (0);
    if sym ≠ period then error (9);
99: writeln
end .
```

5.9. RECOVERING FROM SYNTACTIC ERRORS

Up to this point the parser had only the modest task of determining whether or not an input sequence of symbols belonged to a language. As a side product, the parser also discovered the inherent structure of a sentence. But as soon as an ill-formed construct was encountered, the parser's task was achieved, and the program could as well terminate. For practical compilers, this is of course no tenable proposition. Instead, a compiler must issue an appropriate error diagnostic and be able to continue the parsing process—probably to find further mistakes. A continuation is only possible either by making some likely assumption about the nature of the error and the intention of the author of the ill-formed program or by skipping over some subsequent part of the input sequence, or both. The art of choosing an assumption with a high likelihood of correctness is rather intricate. It has so far eluded any kind of successful formalization because formalizations of syntax and parsing do not take into account the many factors that strongly influence the human mind. For instance, it is a common error to omit interpunctuation symbols such as the semicolon (not only in programming!), whereas it is highly improbable that one forgets to write a + operator in an arithmetic expression. The semicolon and plus symbol are merely terminal symbols without further distinction for the parser; for the human programmer, the semicolon has hardly a meaning and appears redundant at the end of a line, whereas the significance of an arithmetic operator is obvious beyond doubt. There are many more such considerations that have to go into the design of an adequate recovery system, and they all depend on the individual language and cannot be generalized in the framework of all context-free languages.

Nevertheless, there are some rules and hints that can be postulated and that have validity beyond the scope of a single language such as PL/0. Characteristically, perhaps, they are concerned equally much with the initial conception of a language as with the design of the recovery mechanism of its parser. First of all, it is abundantly clear that sensible recovery is much facilitated, or even made possible, only by a *simple language structure*. In particular, if upon diagnosing an error some part of the subsequent input is to be skipped (ignored), then it is mandatory that the language contains *key words* that are highly unlikely to be misused, and that may therefore serve to bring the parser back into step. PL/0 notably follows this rule: every structured statement begins with an unmistakable keyword such as **begin**, **if**, **while**, and the same holds for declarations; they are headed by **var**, **const**, or **procedure**. We shall therefore call this rule the *keyword rule*.

The second rule concerns the construction of the parser more directly. It is the characteristic of top-down parsing that goals are split up into

subgoals and that parsers call upon other parsers to tackle their subgoals. The second rule specifies that if a parser detects an error, it should not merely refuse to continue and report the happening back to its master parser. Instead, it should itself continue to scan text up to a point where some plausible analysis can be resumed. We shall therefore call this the *don't panic rule*. The programmatic consequence of this rule is that there will be no exit from a parser except through its regular termination point.

A possible strict interpretation of the don't panic rule consists of skipping input text upon detecting an illegal formation up to the next symbol that may correctly follow the currently parsed sentential construct. This implies that every parser know the set of its follow-symbols at the place of its present activation.

In the first refinement (or enrichment) step we shall therefore provide every parsing procedure with an explicit parameter *fsys* that specifies the possible follow-symbols. At the end of each procedure an explicit test is included to verify that the next symbol of the input text is indeed among those follow-symbols (if this condition is not already asserted by the logic of the program).

It would, however, be very shortsighted of us to skip the input text up to the next occurrence of such a follow-symbol under all circumstances. After all, the programmer may have mistakenly omitted exactly one symbol (say a semicolon); ignoring the entire text up to the next follow-symbol may be disastrous. We therefore augment these sets of symbols that terminate a possible skip by keywords that specifically mark the beginning of a construct not to be overlooked. The symbols passed as parameters to the parsing procedures are therefore *stopping symbols* rather than follow-symbols only. We may regard the sets of stopping symbols as being initialized by distinct key symbols and being gradually supplemented by legal follow-symbols upon penetration of the hierarchy of parsing subgoals. For flexibility, a general routine called *test* is introduced to perform the described verification. This procedure (5.17) has three parameters:

1. The set $s1$ of admissible next symbols; if the current symbol is not among them, an error is at hand.
2. A set $s2$ of additional stopping symbols whose presence is definitely an error, but which should in no case be ignored and skipped.
3. The number n of the pertinent error diagnostic.

```
procedure test (s1, s2: symset; n: integer);
begin if ¬(sym in s1) then
      begin error(n); s1 := s1+s2;
            while ¬(sym in s1) do getsym
      end
end
```
(5.17)

Procedure (5.17) may also be conveniently used at the *entrance* of parsing procedures to verify whether or not the current symbol is an admissible initial symbol. This is recommended in all cases in which a parsing procedure X is called unconditionally, such as in the statement

$$\begin{array}{l}\textbf{if } sym = a_1 \textbf{ then } S_1 \textbf{ else} \\ \quad \cdots\cdots \\ \textbf{if } sym = a_n \textbf{ then } S_n \textbf{ else } X\end{array}$$

which is the result of translation of the production

$$A ::= a_1S_1 | \ldots | a_nS_n | X \tag{5.18}$$

In these instances the parameter $s1$ must be equal to the set of initial symbols of X, whereas $s2$ is chosen as the set of the follow-symbols of A (see Table 5.2). The details of this procedure are given in Program 5.5, which represents the enriched version of Program 5.4. For the reader's convenience, the entire parser is listed again, with the exception of initializations of global variables and of the procedure *getsym*, all of which remain unchanged.

The scheme presented so far has the property of trying to recover, to fall back into step, by ignoring one or more symbols in the input text. This is an unfortunate strategy in all cases in which an error is caused by *omission* of a symbol. Experience shows that such errors are virtually restricted to symbols which have merely syntactic functions and do not represent an action. An example is the semicolon in PL/0. The fact that the follow-symbol sets are augmented by certain key words actually causes the parser to stop skipping symbols prematurely, thereby behaving as if a missing symbol had been inserted. This can be seen from the program part that parses compound statements shown in (5.19). It effectively "inserts" missing semicolons in front of key words. The set called *statbegsys* is the set of initial symbols of the construct "statement."

```
if sym = beginsym then
begin getsym;
   statement([semicolon, endsym]+fsys);
   while sym in [semicolon]+statbegsys do
      begin
         if sym = semicolon then getsym else error;
         statement([semicolon, endsym]+fsys)
      end;
   if sym = endsym then getsym else error
end
```
(5.19)

The degree of success with which this program diagnoses syntactic errors and recovers from unusual situations can be estimated by considering the PL/0 program (5.20). The listing represents an output delivered by Program

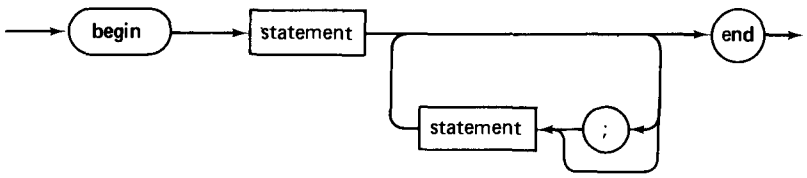


Fig. 5.6 Modified compound statement syntax.

1. Use = instead of :=.
2. = must be followed by a number.
3. Identifier must be followed by =.
4. **const, var, procedure** must be followed by an identifier.
5. Semicolon or comma missing.
6. Incorrect symbol after procedure declaration.
7. Statement expected.
8. Incorrect symbol after statement part in block.
9. Period expected.
10. Semicolon between statements is missing.
11. Undeclared identifier.
12. Assignment to constant or procedure is not allowed.
13. Assignment operator := expected.
14. **call** must be followed by an identifier.
15. Call of a constant or a variable is meaningless.
16. **then** expected.
17. Semicolon or **end** expected.
18. **do** expected.
19. Incorrect symbol following statement.
20. Relational operator expected.
21. Expression must not contain a procedure identifier.
22. Right parenthesis missing.
23. The preceding factor cannot be followed by this symbol.
24. An expression cannot begin with this symbol.
30. This number is too large.

Table 5.3 Error Messages of PL/0 Compiler.

5.5, and Table 5.3 lists a set of possible diagnostic messages corresponding to the error numbers in Program 5.5.

The following program (5.20) was obtained by the introduction of syntactic errors in (5.14) through (5.16).

```
const m = 7, n = 85
var x,y,z,q,r;
   ↑ 5
         ↑ 5
procedure multiply;
   var a,b
```

```
    begin a := u; b := y; z := 0
    ↑ 5
          ↑11
       while b > 0 do
          ↑10
       begin
          if odd b do z := z + a;
                      ↑16
                      ↑19
          a := 2a; b := b/2;
                ↑23
       end
    end ;
procedure divide
    var w;
       ↑ 5
    const two = 2, three := 3;
       ↑ 7
                          ↑ 1
begin r = x; q := 0; w := y;                                    (5.20)
         ↑13
         ↑24
    while w ≤ r do w := two*w;
    while w > y
       begin q := (2*q; w := w/2);
          ↑18
                    ↑22
                                ↑23
          if w ≤ r then
             begin r := r−w q := q+1
                           ↑23
             end
       end
end ;
procedure gcd;
    var f,g;
begin f := x; g := y
    while f ≠ g do
       ↑17
       begin if f < g then g := g−f;
             if g < f then f := f−g;
    z := f
end ;
```

```
begin
   x := m; y := n; call multiply;
   x := 25; y := 3; call divide;
   x := 84; y := 36; call gcd;
   call x; x := gcd; gcd =x
      ↑15
                ↑21
                      ↑12
                        ↑13
                        ↑24
end .
    ↑17
    ↑ 5
    ↑ 7
PROGRAM INCOMPLETE
```

It should be clear that no scheme that reasonably efficiently translates correct sentences will also be able to handle all possible incorrect constructions in a sensible way. And why should it! Every scheme implemented with reasonable effort will fail, that is, will inadequately handle some misconstructions. The important characteristics of a good compiler, however, are that

1. No input sequence will cause the compiler to collapse.
2. All constructs that are illegal according to the language definition are detected and marked.
3. Errors that occur reasonably frequently and are true programmer's mistakes (caused by oversight or misunderstanding) are diagnosed correctly and do not cause any (or many) further stumblings of the compiler (so-called *spurious* error messages).

The presented scheme performs satisfactorily, although there is always room for improvement. Its merit is that it is built according to a few ground rules in a systematic fashion. The ground rules are merely supplemented by some choices of parameters based on heuristics and experience with actual use of the language.

Program 5.5 PL/0 Parser with Error Recovery.

```
program PL0 (input ,output);
{PL/0 compiler with syntax error recovery}
label 99;
const norw = 11;        {no. of reserved words}
   txmax = 100;         {length of identifier table}
   nmax = 14;           {max. no. of digits in numbers}
   al = 10;             {length of identifiers}
type symbol =
   (nul, ident, number, plus, minus, times, slash, oddsym,
   eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
   period, becomes, beginsym, endsym, ifsym, thensym,
   whilesym, dosym, callsym, constsym, varsym, procsym);
   alfa = packed array [1 . . al] of char;
   object = (constant, variable, procedure);
   symset = set of symbol;
var ch: char;           {last character read}
   sym: symbol;         {last symbol read}
   id: alfa;            {last identifier read}
   num: integer;        {last number read}
   cc: integer;         {character count}
   ll: integer;         {line length}
   kk: integer;
   line: array [1 . . 81] of char;
   a: alfa;
   word: array [1 . . norw] of alfa;
   wsym: array [1 . . norw] of symbol;
   ssym: array [char] of symbol;
   declbegsys, statbegsys, facbegsys: symset;
   table: array [0 . . txmax] of
          record name: alfa;
                 kind: object
          end ;
procedure error (n: integer);
begin writeln(' ':cc, '↑', n: 2);
end {error} ;

procedure test (s1,s2: symset; n: integer);
begin if ¬(sym in s1) then
       begin error(n); s1 := s1 + s2;
          while ¬(sym in s1) do getsym
       end
end {test} ;
```

```
procedure block (tx: integer; fsys: symset);
    procedure enter (k: object);
    begin {enter object into table}
        tx := tx + 1;
        with table[tx] do
        begin name := id; kind := k;
        end
    end {enter} ;
    function position (id: alfa): integer;
        var i: integer;
    begin {find identifier id in table}
        table[0] .name := id; i := tx;
        while table[i] .name ≠ id do i := i-1;
        position := i
    end {position} ;
    procedure constdeclaration;
    begin if sym = ident then
        begin getsym;
            if sym in [eql, becomes] then
            begin if sym = becomes then error (1);
                getsym;
                if sym = number then
                    begin enter (constant); getsym
                    end
                else error (2)
            end else error (3)
        end else error (4)
        end {constdeclaration} ;
    procedure vardeclaration;
    begin if sym = ident then
            begin enter (variable); getsym
            end else error (4)
    end {vardeclaration} ;
procedure statement (fsys: symset);
var i: integer;
procedure expression (fsys: symset);
    procedure term (fsys: symset);
        procedure factor (fsys: symset);
            var i: integer;
```

Program 5.5 (Continued)

```
        begin test (facbegsys, fsys, 24);
            while sym in facbegsys do
            begin
                if sym = ident then
                begin i := position (id);
                    if i = 0 then error (11) else
                    if table[i] .kind = procedure then error (21);
                    getsym
                end else
                if sym = number then
                begin getsym;
                end else
                if sym = lparen then
                begin getsym; expression ([rparen]+fsys);
                    if sym = rparen then getsym else error (22)
                end ;
                test(fsys, [lparen], 23)
            end
        end {factor} ;
    begin {term} factor (fsys+[times, slash]);
        while sym in [times, slash] do
            begin getsym; factor(fsys+[times, slash])
            end
    end {term} ;
begin {expression}
    if sym in [plus ,minus] then
        begin getsym; term(fsys+[plus, minus])
        end else term(fsys+[plus, minus]);
    while sym in [plus, minus] do
        begin getsym; term(fsys+[plus, minus])
        end
end {expression} ;
procedure condition(fsys: symset);
begin
    if sym = oddsym then
    begin getsym; expression(fsys);
    end else
    begin expression ([eql, neq, lss, gtr, leq, geq]+fsys);
        if ¬(sym in [eql, neq, lss, leq, gtr, geq]) then
            error (20) else
        begin getsym; expression (fsys)
        end
    end
end {condition} ;
```

Program 5.5 (Continued)

```
begin {statement}
    if sym = ident then
    begin i := position(id);
        if i = 0 then error (11) else
        if table[i] .kind ≠ variable then error (12);
         getsym; if sym = becomes then getsym else error (13);
        expression(fsys);
    end else
    if sym = callsym then
    begin getsym;
        if sym ≠ ident then error (14) else
            begin i := position(id);
                if i = 0 then error (11) else
                if table[i] .kind ≠ procedure then error (15);
                getsym
            end
    end else
    if sym = ifsym then
    begin getsym; condition ([thensym, dosym]+fsys);
        if sym = thensym then getsym else error (16);
        statement(fsys)
    end else
    if sym = beginsym then
    begin getsym; statement([semicolon, endsym]+fsys);
        while sym in [semicolon]+statbegsys do
        begin
            if sym = semicolon then getsym else error (10);
            statement([semicolon ,endsym]+fsys)
        end ;
        if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin getsym; condition([dosym]+fsys);
        if sym = dosym then getsym else error (18);
        statement(fsys);
    end ;
    test(fsys, [ ], 19)
end {statement} ;
```

Program 5.5 (Continued)

```
begin {block}
   repeat
      if sym = constsym then
      begin getsym;
         repeat constdeclaration;
            while sym = comma do
               begin getsym; constdeclaration
               end ;
            if sym = semicolon then getsym else error (5)
         until sym ≠ ident
      end ;
      if sym = varsym then
      begin getsym;
         repeat vardeclaration;
            while sym = comma do
               begin getsym; vardeclaration
               end ;
            if sym = semicolon then getsym else error (5)
         until sym ≠ ident;
      end ;
      while sym = procsym do
      begin getsym;
         if sym = ident then
            begin enter (procedure); getsym
            end
         else error (4);
         if sym = semicolon then getsym else error (5);
         block (tx, [semicolon]+fsys);
         if sym = semicolon then
            begin getsym; test(statbegsys+[ident, procsym], fsys, 6)
            end
         else error (5)
      end ;
      test(statbegsys+[ident], declbegsys, 7)
   until ¬(sym in declbegsys);
   statement([semicolon, endsym]+fsys);
   test(fsys, [ ], 8);
end {block} ;
begin {main program}
   ... Initialization (see Program 5.4) ...
   cc := 0; ll := 0; ch := ' '; kk := al; getsym;
   block (0, [period]+declbegsys+statbegsys);
   if sym ≠ period then error (9);
99: writeln
end .
```

5.10. A PL/0 PROCESSOR

It is indeed remarkable that the PL/0 compiler was so far developed without any knowledge of the machine for which it was supposed to generate code. But why should the structure of an object machine influence the parsing and error recovery scheme of a compiler! In fact, it *must not* do so. Instead, the proper scheme for code generation for any computer should be superimposed on the existing parser by the method of stepwise refinement of the existing program. Since we are about to do this, it becomes necessary to select a processor for which to compile.

In order to keep the description of the compiler reasonably simple and free from extraneous considerations of peculiar properties of a real, existing processor, we shall postulate a computer of our own choice, specifically tailored to the needs of PL/0. Since this processor does not really exist (in hardware), it is a hypothetical processor; it will be called the *PL/0 machine.*

It is not the aim of this section to explain the detailed reasoning that led to the choice of exactly this kind of machine architecture. Instead, it is to serve as a descriptive manual consisting of an intuitive introduction, followed by a detailed definition of the processor in the form of an algorithm. This formalization may serve as an example for accurate and detailed algorithmic descriptions of actual processors. The algorithm interprets PL/0 instructions sequentially, and is called an *interpreter.*

The PL/0 machine consists of two stores: an instruction register and three address registers. The *program store*, called *code*, is loaded by the compiler and remains unchanged during interpretation of the code. It can then be considered as a read-only store. The *data store S* is organized as a *stack*, and all arithmetic operators operate on the two elements on top of the stack, replacing their operands by a result. The top element is addressed (indexed) by the *top stack register T.* The *instruction register I* contains the instruction that is currently being interpreted. The *program address register P* designates the next instruction to be fetched for interpretation.

Every procedure in PL/0 may contain local variables. Since procedures may be activated recursively, storage for these local variables may not be allocated before the actual procedure call. Hence, the data segments for individual procedures are stacked up consecutively in the stack store *S.* Since procedure activations strictly obey the first-in-last-out scheme, the stack is the appropriate storage allocation strategy. Every procedure owns some internal information of its own, namely, the program address of its call (the so-called *return address*), and the address of the data segment of its caller. These two addresses are needed for proper resumption of program execution after termination of the procedure. They can be understood as internal or implicit local variables allocated in the procedure's data segment.

We call them the *return address RA* and the *dynamic link DL*. The origin of the dynamic link, that is, the address of the most recently allocated data segment, is retained in the *base address register B*.

Since the actual allocation of storage takes place during execution (interpretation) time, the compiler cannot equip the generated code with absolute addresses. Since it can only determine the location of variables within a data segment, it is capable of providing *relative addresses* only. The interpreter has to add to this so-called *displacement* to the base address of the appropriate data segment. If a variable is local to the procedure currently being interpreted, then this base address is given by the *B* register. Otherwise, it must be obtained by descending the chain of data segments. The compiler, however, can only know the static depth of an access path, whereas the dynamic link chain maintains the dynamic history of procedure activations. Unfortunately, these two access paths are not necessarily the same.

For example, assume that a procedure *A* calls a procedure *B* declared local to *A*, *B* calls *C* declared local to *B*, and *C* calls *B* (recursively). We say that *A* is declared at level 1, *B* at level 2, *C* at level 3 (see Fig. 5.7). If a variable *a* declared in *A* is to be accessed in *B*, then the compiler knows that there exists a *level difference* of 1 between *B* and *A*. Descending one step along the dynamic link chain, however, would result in an access to a variable local to *C*!

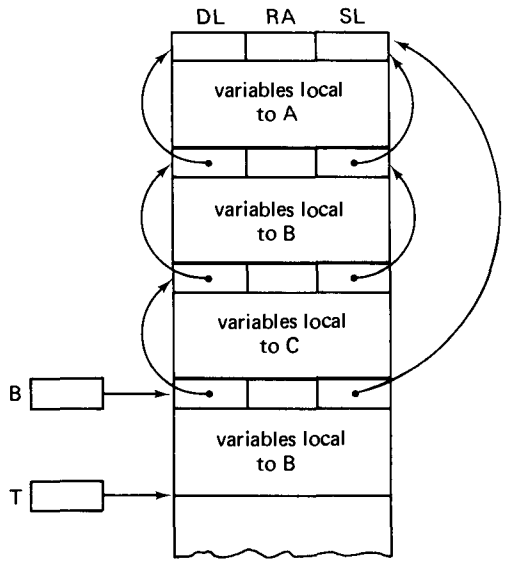


Fig. 5.7 Stack of PL/0 machine.

Hence, it is plain that a second link chain has to be provided that properly links data segments in the way the compiler can see the situation. We call this the *static link SL*.

Addresses are therefore generated as pairs of numbers indicating the static level difference and the relative displacement within a data segment. We assume that each location of the data store is capable of holding an address or an integer.

The instruction set of the PL/0 machine is tuned to the requirements of the PL/0 language. It includes the following orders:

1. An instruction to load numbers (literals) onto the stack (LIT).
2. An instruction to fetch variables onto the top of the stack (LOD).
3. A store instruction corresponding to assignment statements (STO).
4. An introduction to activate a subroutine corresponding to a procedure call (CAL).
5. An instruction to allocate storage on the stack by incrementing the stack pointer T (INT).
6. Instructions for unconditional and conditional transfer of control, used in if- and while statements (JMP, JPC).
7. A set of arithmetic and relational operators (OPR).

The format of instructions is determined by the need for three components, namely, an operation code f and a parameter consisting of one or two parts (see Fig. 5.8). In the case of operators the parameter a determines the identity of the operator; in the other cases it is either a number (LIT, INT), a program address (JMP, JPC, CAL), or a data address (LOD, STO).

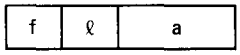


Fig. 5.8 Instruction format.

The details of operation of the PL/0 machine should be evident from the procedure called *interpret* that is part of Program 5.6, which combines the completed compiler with the interpreter into a system that translates and subsequently executes PL/0 programs. The modification of this program to generate code for an existing computer is left as an excercise for the interested reader. The resulting expansion of the compiler program may be taken as a measure of the appropriateness of the chosen computer for the present task.

There is no doubt that the presented PL/0 computer could be expanded into a more sophisticated organization in order to make certain operations more efficient. One instance is the chosen addressing mechanism. The presented solution was chosen because of its inherent simplicity and because all improvements must essentially be based on it and derived from it.

5.11. CODE GENERATION

In order to be able to assemble an instruction, the compiler must know its operation code and its parameter, which is a literal number or an address. These values are associated by the compiler itself with the respective identifiers. This association is performed upon processing the declaration of constants, variables, and procedures. For this purpose, the table containing the identifiers is expanded to contain the attributes associated with each identifier. If an identifier denotes a constant, its attribute is the constant value; if the identifier denotes a variable, the attribute is its address, consisting of a displacement and a level; and if the identifier denotes a procedure, then its attributes are the procedure's entry address and its level. The corresponding extension of the declaration of the variable *table* is shown in Program 5.6. It is a noteworthy example of a stepwise refinement (or enrichment) of a data declaration progressing simultaneously with the refinement of the statement part.

Whereas the constant values are provided by the program text, it is the compiler's task to determine addresses on its own. PL/0 is sufficiently simple to make sequential allocation of variables and code the obvious choice. Hence, every variable declaration is processed by incrementing a data allocation index by 1 (since each variable occupies by definition of the PL/0 machine exactly one storage cell). The data allocation index *dx* is to be initialized upon starting the compilation of a procedure, reflecting the fact that its data segment starts empty. [Actually, *dx* is given the initial value 3 since each data segment contains at least the three internal variables *RA*, *DL*, *SL* (see preceding section).] The appropriate computations to determine the identifiers' attributes are included in the procedure *enter* which is used to enter new identifiers into the table.

With this information about operands at hand, generating the actual code is a rather simple affair. Because of the convenient stack organization of the PL/0 machine, there exists practically a one-to-one correspondence between operands and operators in the source language and instructions in the target code. The compiler has merely to perform a suitable resequencing into *postfix* form. By "postfix form" is meant that operators always follow their operands instead of being embedded between the operands as in the conventional *infix* form. The postfix form is sometimes also called Polish form (after its originator Lukasciewicz) or *parenthesis-free* form since it makes parentheses superfluous. Some correspondences between infix and postfix forms of expressions are shown in Table 5.4 (see also Sect. 4.4.2).

The very simple technique of performing this transformation is shown by the procedures *expression* and *term* in Program 5.6. It is merely a matter

Infix Form	Postfix Form
$x + y$	$xy+$
$(x - y) + z$	$xy - z+$
$x - (y + z)$	$xyz+-$
$x*(y + z)*w$	$xyz + *w*$

Table 5.4 Expressions in Infix and Postfix Form.

of delaying the transmission of the arithmetic operator. At this point the reader should verify that the presented arrangement of parsing procedures also takes care of an appropriate interpretation of the conventional priority rules among the various operators.

A slightly less trivial matter is the translation of conditional and repetitive statements. In this case the generation of jump instructions is necessary, for which at times the destination address is still unknown. If one insists on a strictly sequential production of instructions in the form of an output file, then a *two-pass* compiler scheme is necessary. The second pass then assumes the task of supplementing the incomplete jump instructions with their destination addresses. An alternative solution adopted by the present compiler is to place the instructions into an array and essentially retaining them in directly accessible store. This method allows supplementing the missing addresses as soon as they become known. This operation is commonly called a *fixup*.

The only additional operation that has to be performed when issuing such a forward jump is to retain its location, i.e., its index in the program store. This address is then used to locate the incomplete instruction at the time of the fixup. The details are again evident from Program 5.6 (see routines processing **if**- and **while** statements). The patterns of code generated for the **if**- and **while** statements are as follows ($L1$ and $L2$ stand for code addresses):

if C **then** S	**while** C **do** S
code for condition C	L1: code for C
JPC L1	JPC L2
code for statement S	code for S
L1: ...	JMP L1
	L2: ...

For convenience, an auxiliary procedure called *gen* is introduced. Its purpose is to assemble and emit an instruction according to its three parameters. It automatically increments the code index *cx* which designates the location of the next instruction to be issued.

As an example, the code emitted by compiling the multiplication routine (5.14) is listed below in mnemonic form. The comments on the right-hand side are merely added for explanatory purposes.

```
 2   INT   0,5    allocate space for links and local variables
 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
```

Code corresponding to PL/0 procedure 5.14.

Many tasks in compiling programming languages are considerably more complex than the ones presented in the PL/0 compiler for the PL/0 machine [5-4]. Most of them are much more resistant to being neatly organized. The reader trying to extend the presented compiler in either direction toward a more powerful language or a more conventional computer will soon realize the truth of this statement. Nevertheless, the basic approach toward designing a complex program presented here retains its validity, and even increases its value when the task grows more complicated and more sophisticated. It has, in fact, been successfully used in the construction of large compilers [5-1 and 5-9].

Program 5.6 PL/0 Compiler.

```
program PL0(input,output);
{PL/0 compiler with code generation}
label 99;
const norw = 11;      {no. of reserved words}
   txmax = 100;       {length of identifier table}
   nmax = 14;         {max. no. of digits in numbers}
   al = 10;           {length of identifiers}
   amax = 2047;       {maximum address}
   levmax = 3;        {maximum depth of block nesting}
   cxmax = 200;       {size of code array}
type symbol =
   (nul, ident, number, plus, minus, times, slash, oddsym,
    eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
    period, becomes, beginsym, endsym, ifsym, thensym,
    whilesym, dosym, callsym, constsym, varsym, procsym);
   alfa = packed array [1 .. al] of char;
   object = (constant, variable, procedure);
   symset = set of symbol;
   fct = (lit, opr, lod, sto, cal, int, jmp, jpc);      {functions}
   instruction = packed record
                    f: fct;          {function code}
                    l: 0 .. levmax;  {level}
                    a: 0 .. amax;    {displacement address}
                 end ;
{  LIT 0,a  :  load constant a
   OPR 0,a  :  execute operation a
   LOD l,a  :  load variable l,a
   STO l,a  :  store variable l,a
   CAL l,a  :  call procedure a at level l
   INT 0,a  :  increment t-register by a
   JMP 0,a  :  jump to a
   JPC 0,a  :  jump conditional to a    }
var ch: char;          {last character read}
   sym: symbol;        {last symbol read}
   id: alfa;           {last identifier read}
   num: integer;       {last number read}
   cc: integer;        {character count}
   ll: integer;        {line length}
   kk, err: integer;
   cx: integer;        {code allocation index}
```

```
    line: array [1 . . 81] of char;
    a: alfa;
    code: array [0 . . cxmax] of instruction;
    word: array [1 . . norw] of alfa;
    wsym: array [1 . . norw] of symbol;
    ssym: array [char] of symbol;
    mnemonic: array [fct] of
                  packed array [1 . . 5] of char;
    declbegsys, statbegsys, facbegsys: symset;
    table: array [0 . . txmax] of
        record name: alfa;
            case kind: object of
            constant: (val: integer);
            variable, procedure: (level, adr: integer)
        end ;
procedure error(n: integer);
begin writeln(' ****', ' ': cc-1, '↑', n: 2); err := err+1
end {error} ;

procedure getsym;
    var i,j,k: integer;

    procedure getch;
    begin if cc = ll then
        begin if eof(input) then
                  begin write(' PROGRAM INCOMPLETE'); goto 99
                  end ;
            ll := 0; cc := 0; write(cx: 5, ' ');
            while ¬eoln(input) do
                begin ll := ll+1; read(ch); write(ch); line[ll] := ch
                end ;
            writeln; ll := ll+1; read(line[ll])
        end ;
        cc := cc+1; ch := line[cc]
    end {getch} ;

begin {getsym}
    while ch = ' ' do getch;
    if ch in ['A' . . 'Z'] then
    begin {identifier or reserved word} k := 0;
        repeat if k < al then
            begin k := k+1; a[k] := ch
            end ;
```

Program 5.6 (Continued)

```
            getch
        until ¬(ch in ['A'..'Z','0'..'9']);
        if k ≥ kk then kk := k else
            repeat a[kk] := ' '; kk := kk-1
            until kk = k;
        id := a; i := 1; j := norw;
        repeat k := (i+j) div 2;
            if id ≤ word[k] then j := k-1;
            if id ≥ word[k] then i := k+1
        until i > j;
        if i-1 > j then sym := wsym[k] else sym := ident
    end else
    if ch in ['0'..'9'] then
    begin {number} k := 0; num := 0; sym := number;
        repeat num := 10*num + (ord(ch)-ord('0'));
            k := k+1; getch
        until ¬(ch in ['0'..'9']);
        if k > nmax then error (30)
    end else
    if ch = ':' then
    begin getch;
        if ch = '=' then
        begin sym := becomes; getch
        end else sym := nul;
    end else
    begin sym := ssym[ch]; getch
    end
end {getsym} ;

procedure gen(x: fct; y,z: integer);
begin if cx > cxmax then
            begin write(' PROGRAM TOO LONG'); goto 99
            end ;
    with code[cx] do
        begin f := x; l := y; a := z
        end ;
    cx := cx + 1
end {gen} ;

procedure test(s1,s2: symset; n: integer);
begin if ¬(sym in s1) then
        begin error(n); s1 := s1 + s2;
            while ¬(sym in s1) do getsym
        end
end {test} ;
```

Program 5.6 (Continued)

```
procedure block(lev,tx: integer; fsys: symset);
   var dx: integer;      {data allocation index}
      tx0: integer;      {initial table index}
      cx0: integer;      {initial code index}
   procedure enter(k: object);
   begin {enter object into table}
      tx := tx + 1;
      with table[tx] do
      begin name := id; kind := k;
         case k of
         constant: begin if num > amax then
                                 begin error (30); num := 0 end ;
                     val := num
                   end ;
         variable: begin level := lev; adr := dx; dx := dx+1;
                   end ;
         procedure: level := lev
         end
      end
   end {enter} ;
   function position(id: alfa): integer;
      var i: integer;
   begin {find indentifier id in table}
      table[0] .name := id; i := tx;
      while table[i] .name ≠ id do i := i-1;
      position := i
   end {position} ;
   procedure constdeclaration;
   begin if sym = ident then
      begin getsym;
         if sym in [eql ,becomes] then
         begin if sym = becomes then error(1);
            getsym;
            if sym = number then
               begin enter(constant); getsym
               end
            else error (2)
         end else error (3)
      end else error (4)
   end {constdeclaration} ;
procedure vardeclaration;
begin if sym = ident then
      begin enter(variable); getsym
      end else error (4)
end {vardeclaration} ;
```

Program 5.6 (Continued)

```
procedure listcode;
    var i: integer;
begin {list code generated for this block}
    for i := cx0 to cx-1 do
        with code[i] do
            writeln(i, mnemonic[f]:5, l:3, a:5)
end {listcode} ;

procedure statement(fsys: symset);
    var i,cx1,cx2: integer;
    procedure expression(fsys: symset);
        var addop: symbol;
        procedure term(fsys: symset);
            var mulop: symbol;
            procedure factor(fsys: symset);
                var i: integer;
            begin test(facbegsys, fsys, 24);
                while sym in facbegsys do
                begin
                    if sym = ident then
                    begin i := position(id);
                        if i = 0 then error (11) else
                        with table[i] do
                        case kind of
                          constant: gen(lit, 0, val);
                          variable: gen(lod, lev-level, adr);
                          procedure: error (21)
                        end ;
                        getsym
                    end else
                    if sym = number then
                    begin if num > amax then
                            begin error (30); num := 0
                            end ;
                        gen(lit, 0, num); getsym
                    end else
                    if sym = lparen then
                    begin getsym; expression([rparen]+fsys);
                        if sym = rparen then getsym else error (22).
                    end ;
                    test(fsys, [lparen], 23)
                end
            end {factor} ;
```

Program 5.6 (Continued)

```
    begin {term} factor(fsys+[times, slash]);
        while sym in [times, slash] do
            begin mulop := sym; getsym; factor(fsys+[times, slash]);
                if mulop = times then gen(opr,0,4) else gen(opr,0,5)
            end
    end {term} ;
begin {expression}
    if sym in [plus, minus] then
        begin addop := sym; getsym; term(fsys+[plus, minus]);
            if addop = minus then gen(opr,0,1)
        end else term(fsys+[plus, minus]);
    while sym in [plus, minus] do
        begin addop := sym; getsym; term(fsys+[plus, minus]);
            if addop = plus then gen(opr,0,2) else gen(opr,0,3)
        end
end {expression} ;

procedure condition(fsys: symset);
    var relop: symbol;
begin
    if sym = oddsym then
    begin getsym; expression(fsys); gen(opr,0,6)
    end else
    begin expression([eql, neq, lss, gtr, leq, geq]+fsys);
        if ¬(sym in [eql, neq, lss, leq, gtr, geq]) then
            error (20) else
        begin relop := sym; getsym; expression(fsys);
            case relop of
              eql: gen(opr,0, 8);
              neq: gen(opr,0, 9);
              lss: gen(opr,0,10);
              geq: gen(opr,0,11);
              gtr: gen(opr,0,12);
              leq: gen(opr,0,13);
            end
        end
    end
end {condition} ;
```

Program 5.6 (Continued)

```
begin {statement}
    if sym = ident then
    begin i := position(id);
        if i = 0 then error (11) else
        if table[i] .kind ≠ variable then
            begin {assignment to non-variable} error (12); i := 0
            end ;
        getsym; if sym = becomes then getsym else error (13);
        expression(fsys);
        if i ≠ 0 then
            with table[i] do gen(sto, lev-level, adr)
    end else
    if sym = callsym then
    begin getsym;
        if sym ≠ ident then error (14) else
            begin i := position(id);
                if i = 0 then error (11) else
                with table[i] do
                    if kind = procedure then gen (cal, lev-level, adr)
                    else error (15);
                getsym
            end
    end else
    if sym = ifsym then
    begin getsym; condition([thensym, dosym]+fsys);
        if sym = thensym then getsym else error (16);
        cx1 := cx; gen(jpc,0,0);
        statement(fsys); code[cx1].a := cx
    end else
    if sym = beginsym then
    begin getsym; statement([semicolon, endsym]+fsys);
        while sym in [semicolon]+statbegsys do
        begin
            if sym = semicolon then getsym else error (10);
            statement([semicolon, endsym]+fsys)
        end ;
        if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin cx1 := cx; getsym; condition([dosym]+fsys);
        cx2 := cx; gen(jpc,0,0);
        if sym = dosym then getsym else error (18);
        statement(fsys); gen(jmp,0,cx1); code[cx2].a := cx
    end ;
    test(fsys, [ ], 19)
end {statement} ;
```

```
begin {block} dx := 3; tx0 := tx; table[tx] .adr := cx; gen(jmp,0,0);
    if lev > levmax then error (32);
    repeat
        if sym = constsym then
        begin getsym;
            repeat constdeclaration;
                while sym = comma do
                    begin getsym; constdeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident
        end ;
        if sym = varsym then
        begin getsym;
            repeat vardeclaration;
                while sym = comma do
                    begin getsym; vardeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident;
        end ;
        while sym = procsym do
        begin getsym;
            if sym = ident then
                begin enter(procedure); getsym
                end
            else error (4);
            if sym = semicolon then getsym else error (5);
            block(lev+1,tx,[semicolon]+fsys);
            if sym = semicolon then
                begin getsym; test(statbegsys+[ident, procsym], fsys, 6)
                end
            else error (5)
        end ;
        test(statbegsys+[ident], declbegsys, 7)
    until ¬(sym in declbegsys);
    code[table[tx0].adr].a := cx;
    with table[tx0] do
        begin adr := cx; {start adr of code}
        end ;
    cx0 := cx; gen(int,0,dx);
    statement([semicolon, endsym]+fsys);
    gen(opr,0,0); {return}
    test(fsys, [ ], 8);
    listcode;
end {block} ;
```

```
procedure interpret;
    const stacksize = 500;
    var p,b,t: integer {program-, base-, topstack-registers}
        i: instruction; {instruction register}
        s: array [1 . . stacksize] of integer;  {datastore}
    function base(l: integer): integer;
        var b1: integer;
    begin b1 := b; {find base l levels down}
        while l > 0 do
            begin b1 := s[b1]; l := l-1
            end ;
        base := b1
    end {base} ;

begin writeln(' START PL/0');
    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
      lit: begin t := t+1; s[t] := a
           end ;
      opr: case a of     {operator}
           0: begin {return}
                  t := b-1; p := s[t+3]; b := s[t+2];
              end ;
           1: s[t] := -s[t];
           2: begin t := t-1; s[t] := s[t] + s[t+1]
              end ;
           3: begin t := t-1; s[t] := s[t] - s[t+1]
              end ;
           4: begin t := t-1; s[t] := s[t] * s[t+1]
              end ;
           5: begin t := t-1; s[t] := s[t] div s[t+1]
              end ;
           6: s[t] := ord(odd(s[t]));
           8: begin t := t-1; s[t] := ord(s[t]=s[t+1])
              end ;
           9: begin t := t-1; s[t] := ord(s[t]≠s[t+1])
              end ;
          10: begin t := t-1; s[t] := ord(s[t]<s[t+1])
              end ;
          11: begin t := t-1; s[t] := ord(s[t]≥s[t+1])
              end ;
```

Program 5.6 (Continued)

```
            12: begin t := t-1; s[t] := ord(s[t]>s[t+1])
                end ;
            13: begin t := t-1; s[t] := ord(s[t]≤s[t+1])
                end ;
          end ;
      lod: begin t := t+1; s[t] := s[base(l)+a]
          end ;
      sto: begin s[base(l)+a] := s[t]; writeln(s[t]); t := t-1
          end ;
      cal: begin {generate new block mark}
                s[t+1] := base(l); s[t+2] := b; s[t+3] := p;
                b := t+1; p := a
           end ;
      int: t := t+a;
      jmp: p := a;
      jpc: begin if s[t] = 0 then p := a; t := t-1
          end
        end {with, case}
    until p = 0;
    write(' END PL/0');
end {interpret} ;

begin {main program}
    for ch := 'A' to ';' do ssym[ch] := nul;
    word[ 1] := 'BEGIN ';   word[ 2] := 'CALL      ';
    word[ 3] := 'CONST';    word[ 4] := 'DO        ';
    word[ 5] := 'END   ';   word[ 6] := 'IF        ';
    word[ 7] := 'ODD   ';   word[ 8] := 'PROCEDURE';
    word[ 9] := 'THEN  ';   word[10] := 'VAR       ';
    word[11] := 'WHILE ';
    wsym[ 1] := beginsym; wsym[ 2] := callsym;
    wsym[ 3] := constsym; wsym[ 4] := dosym;
    wsym[ 5] := endsym;   wsym[ 6] := ifsym;
    wsym[ 7] := oddsym;   wsym[ 8] := procsym;
    wsym[ 9] := thensym;  wsym[10] := varsym;
    wsym[11] := whilesym;
    ssym['+'] := plus;     ssym['-'] := minus;
    ssym['*'] := times;    ssym['/'] := slash;
    ssym['('] := lparen;   ssym[')'] := rparen;
    ssym['='] := eql;      ssym[','] := comma;
    ssym['.'] := period;   ssym['≠'] := neq;
    ssym['<'] := lss;      ssym['>'] := gtr;
    ssym['≤'] := leq;      ssym['≥'] := geq;
    ssym[';'] := semicolon;
```

Program 5.6 (Continued)

```
    mnemonic[lit] := 'LIT '; mnemonic[opr] := 'OPR';
    mnemonic[lod] := 'LOD'; mnemonic[sto] := 'STO';
    mnemonic[cal] := 'CAL'; mnemonic[int] := 'INT ';
    mnemonic[jmp] := 'JMP'; mnemonic[jpc] := 'JPC';
    declbegsys := [constsym, varsym, procsym];
    statbegsys := [beginsym, callsym, ifsym, whilesym];
    facbegsys := [ident, number, lparen];
    page(output); err := 0
    cc := 0; cx := 0; ll := 0; ch := ' '; kk := al; getsym;
    block(0,0,[period]+declbegsys+statbegsys);
    if sym ≠ period then error (9);
    if err = 0 then interpret else write(' ERRORS IN PL/0 PROGRAM');
99: writeln
end .
```

Program 5.6 (Continued)

EXERCISES

5.1. Consider the following syntax.

$$
\begin{aligned}
S &::= A \\
A &::= B \mid \textbf{if } A \textbf{ then } A \textbf{ else } A \\
B &::= C \mid B{+}C \mid {+}C \\
C &::= D \mid C{*}D \mid {*}D \\
D &::= x \mid (A) \mid -D
\end{aligned}
$$

Which are the terminal and the non-terminal symbols? Determine the sets of leftmost and follow-symbols $L(X)$ and $F(X)$ for each non-terminal symbol X. Construct a sequence of parsing steps for the following sentences:

$x{+}x$
$(x{+}x){*}({+}-x)$
$(x{*}-{+}x)$
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

5.2. Does the grammar of Exercise 5.1 satisfy the Restrictive Rules 1 and 2 for one-symbol lookahead top-down parsing? If not, find an equivalent syntax which does satisfy these rules. Represent this syntax by a syntax graph and a data structure to be used by Program 5.3.

5.3. Repeat Exercise 5.2 for the following syntax:

$$S ::= A$$
$$A ::= B \mid \textbf{if } C \textbf{ then } A \mid \textbf{if } C \textbf{ then } A \textbf{ else } A$$
$$B ::= D = C$$
$$C ::= \textbf{if } C \textbf{ then } C \textbf{ else } C \mid D$$

Hint: You may find it necessary to delete or replace some construct in order to allow for one-symbol top-down parsing to be applicable.

5.4. Given the following syntax, consider the problem of top-down parsing:

$$S ::= A$$
$$A ::= B{+}A \mid DC$$
$$B ::= D \mid D{*}B$$
$$D ::= x \mid (C)$$
$$C ::= +x \mid -x$$

How many symbols do you have to look ahead at most in order to parse sentences according to this syntax?

5.5. Transform the description of PL/0 (Fig. 5.4) into an equivalent set of BNF-productions.

5.6. Write a program that determines the sets of initial and follow-symbols $L(S)$ and $F(S)$ for each non-terminal symbol S in a given set of productions.

Hint: Use part of Program 5.3 to construct an internal representation of the syntax in the form of a data structure. Then operate on this linked data structure.

5.7. Extend the PL/0 language and its compiler by the following features:

(a) A conditional statement of the form

⟨*statement*⟩ ::= **if** ⟨*condition*⟩ **then** ⟨*statement*⟩ **else** ⟨*statement*⟩

(b) A repetitive statement of the form

⟨*statement*⟩ ::= **repeat** ⟨*statement*⟩ {; ⟨*statement*⟩} **until** ⟨*condition*⟩

Are there any particular difficulties that might cause a change of form or interpretation of the given PL/0 features? You should not introduce any additional instructions in the repertoire of the PL/0 machine.

5.8. Extend the PL/0 language and compiler by introducing procedure parameters. Consider two possible solutions and chose one of them for your realization.

(a) *Value parameters.* The actual parameters in the call are expressions whose values are assigned to local variables represented by the formal parameters specified in the procedure heading.

(b) *Variable parameters.* The actual parameters are variables. Upon a call, they are substituted in place of the formal parameters. Variable parameters are implemented by passing the address of the actual parameter, storing it in the location denoted by the formal parameters. The actual parameters are then accessed indirectly via the transmitted address.

Hence, variable parameters provide access to variables defined outside the procedures, and the rules of scope may therefore be changed as follows: In every procedure only local variables may be accessed directly; non-local variables are accessible exclusively via parameters.

5.9. Extend the PL/0 language and compiler by introducing arrays of variables. Assume that the range of indices of an array variable *a* is indicated in its declaration as

var *a*(*low* : *high*)

5.10. Modify the PL/0 compiler to generate code for your available computer.

Hint: Generate symbolic assembly code in order to avoid problems with loader conventions. In a first step avoid trying to optimize code, for example, with respect to register usage. Possible optimizations should be incorporated in a fourth refinement step of the compiler.

5.11. Extend Program 5.5 into a program called "prettyprint." The purpose of this program is to read PL/0-texts and print them in a layout which naturally reflects the textual structure by appropriate line separation and indentation. First define accurate line separation and indentation rules based on the syntactic structure of PL/0; then implement them by superimposing write statements onto Program 5.5. (Write statements must be removed from the scanner, of course.)

REFERENCES

5-1. AMMANN, U., "The Method of Structured Programming Applied to the Development of a Compiler," *International Computing Symposium 1973*, A. Günther et al. eds., (Amsterdam: North-Holland Publishing Co., 1974), pp. 93–99.

5-2. COHEN, D. J. and GOTLIEB, C. C., "A List Structure Form of Grammars for Syntactic Analysis," *Comp. Surveys*, **2**, No. 1 (1970), 65–82.

5-3. FLOYD, R. W., "The Syntax of Programming Languages—A Survey," *IEEE Trans.*, EC-13 (1964), 346–53.

5-4. GRIES, D., *Compiler Construction for Digital Computers* (New York: Wiley, 1971).

5-5. KNUTH, D. E., "Top-down Syntax Analysis," *Acta Informatica*, **1**, No. 2 (1971), 79–110.

5-6. LEWIS, P. M. and STEARNS, R. E., "Syntax-directed Transduction," *J. ACM*, **15**, No. 3 (1968), 465–88.

5-7. NAUR, P., ed., "Report on the Algorithmic Language ALGOL 60," *ACM*, **6**, No. 1 (1963), 1–17.

5-8. SCHORRE, D. V., "META II, A Syntax-oriented Compiler Writing Language," *Proc. ACM Natl. Conf.*, **19**, (1964), D 1.3.1-11.

5-9. WIRTH, N., "The Design of a PASCAL Compiler," *Software-Practice and Experience*, **1**, No. 4 (1971), 309–33.

A THE ASCII CHARACTER SET

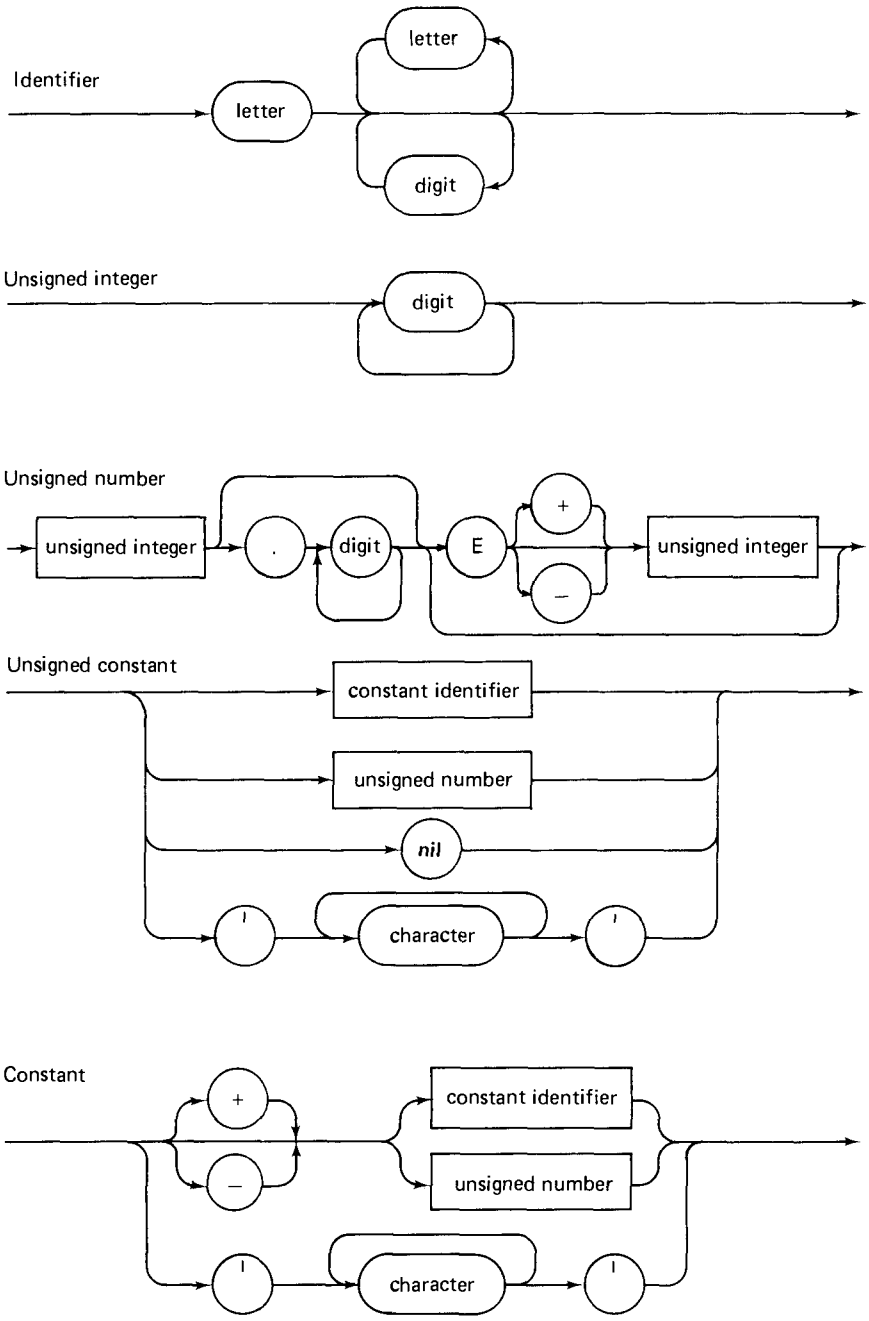
y \ x	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l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

The ordinal number of a character *ch* is computed from its coordinates in the table as

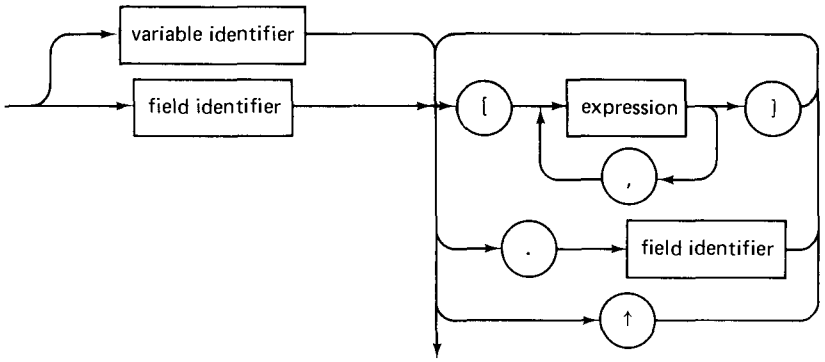
$$ord(ch) = 16*x + y$$

The characters with ordinal numbers 0 through 31 and 127 are so-called *control characters* used for data transmission and device control. The character with ordinal number 32 is the blank.

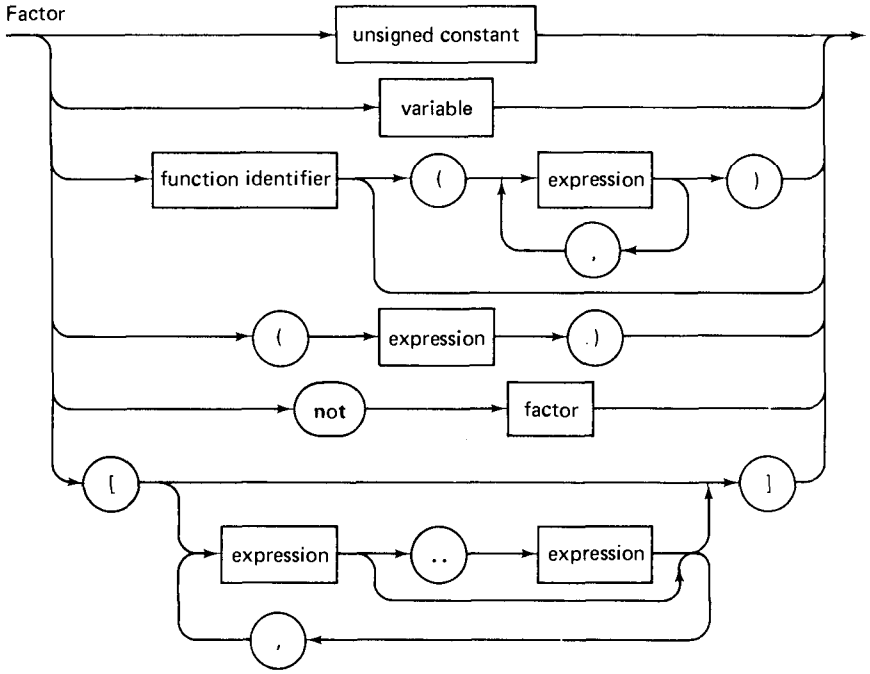
B PASCAL SYNTAX DIAGRAMS


Identifier
letter
letter
digit
Unsigned integer
digit
Unsigned number
unsigned integer
.
digit
E
+
–
unsigned integer
Unsigned constant
constant identifier
unsigned number
nil
'
character
'
Constant
+
–
constant identifier
unsigned number
'
character
'

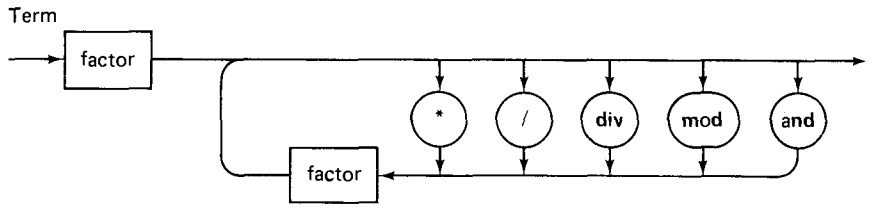
Variable



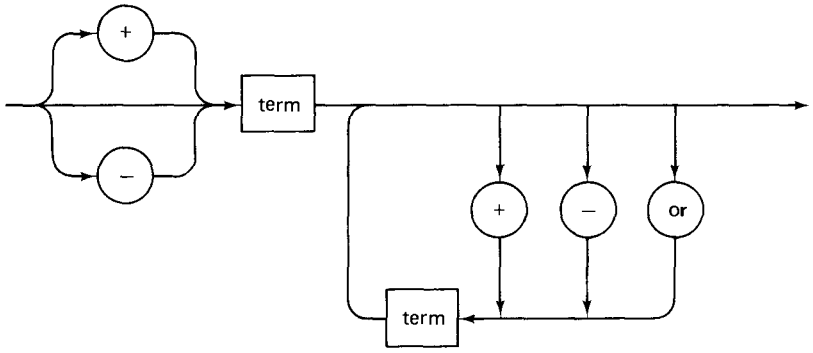
Factor



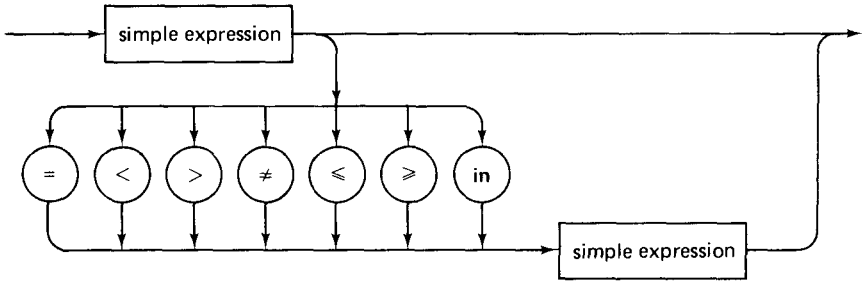
Term



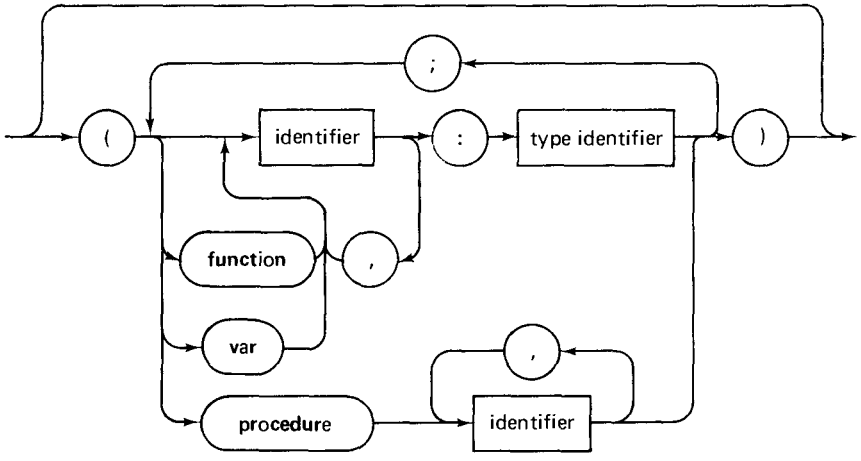
Simple expression



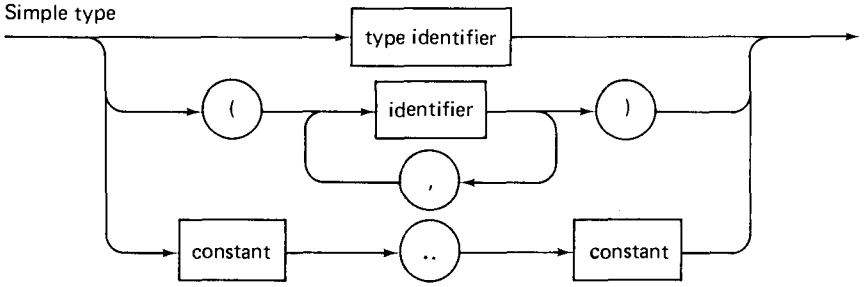
Expression



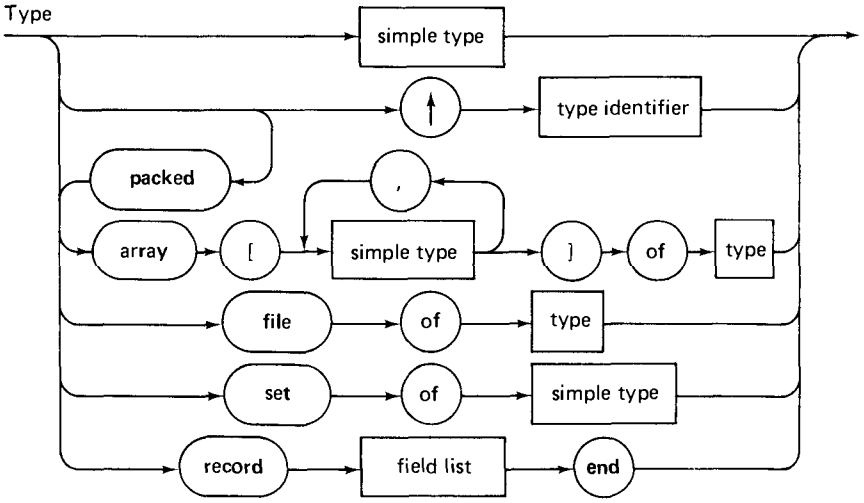
Parameter list



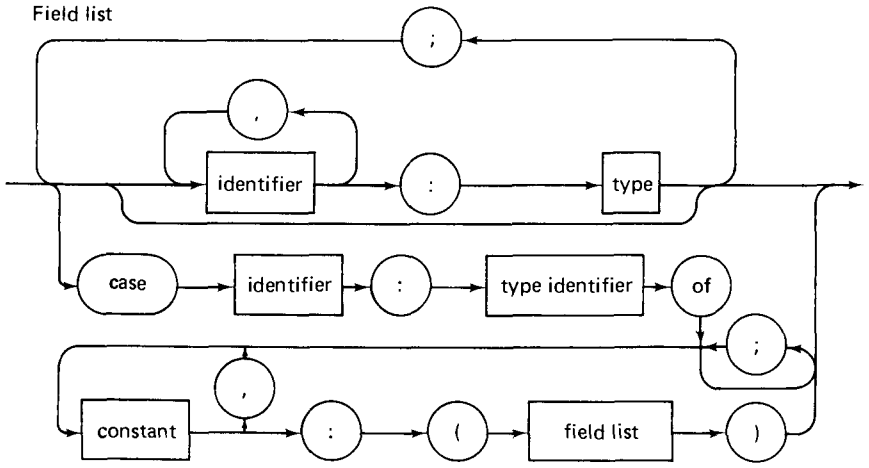
Simple type

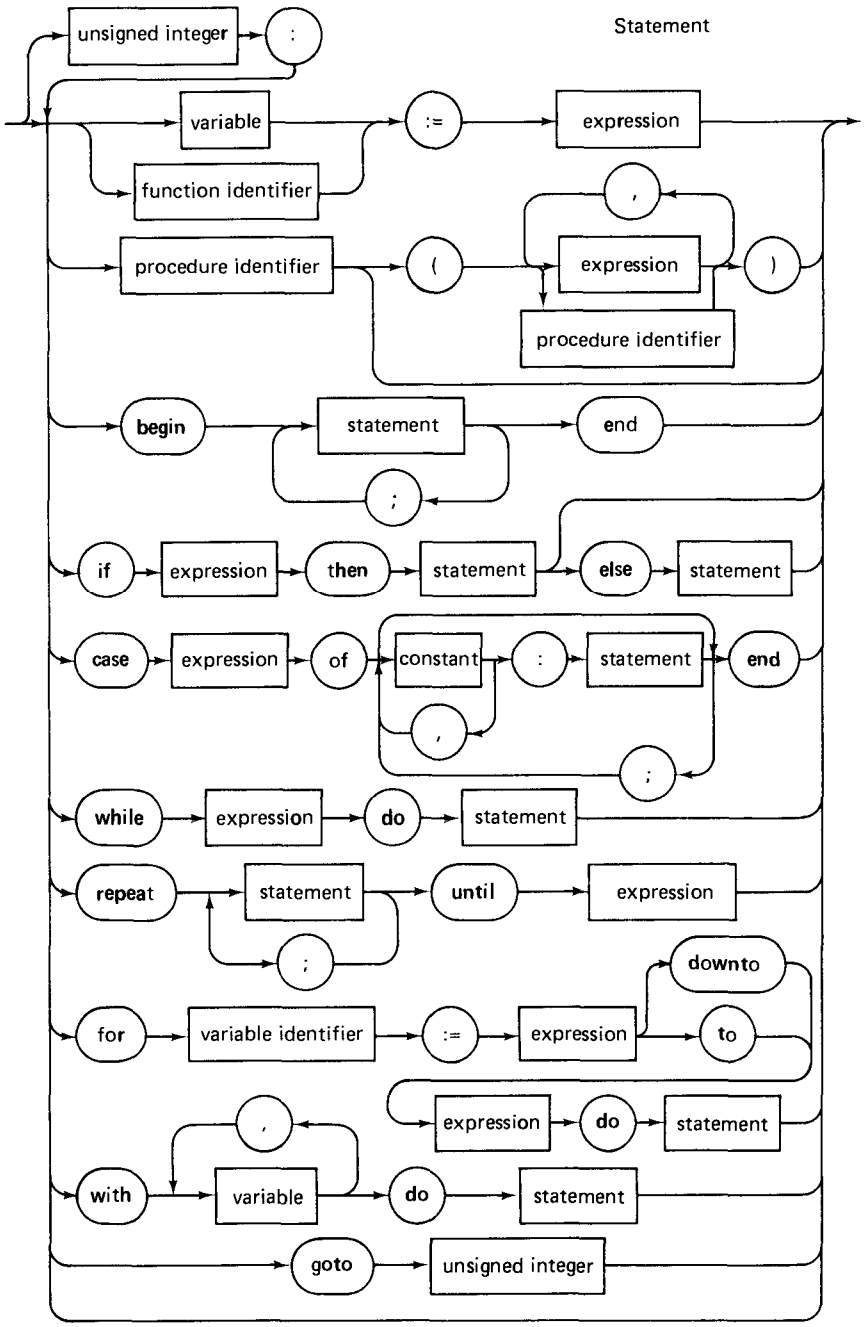


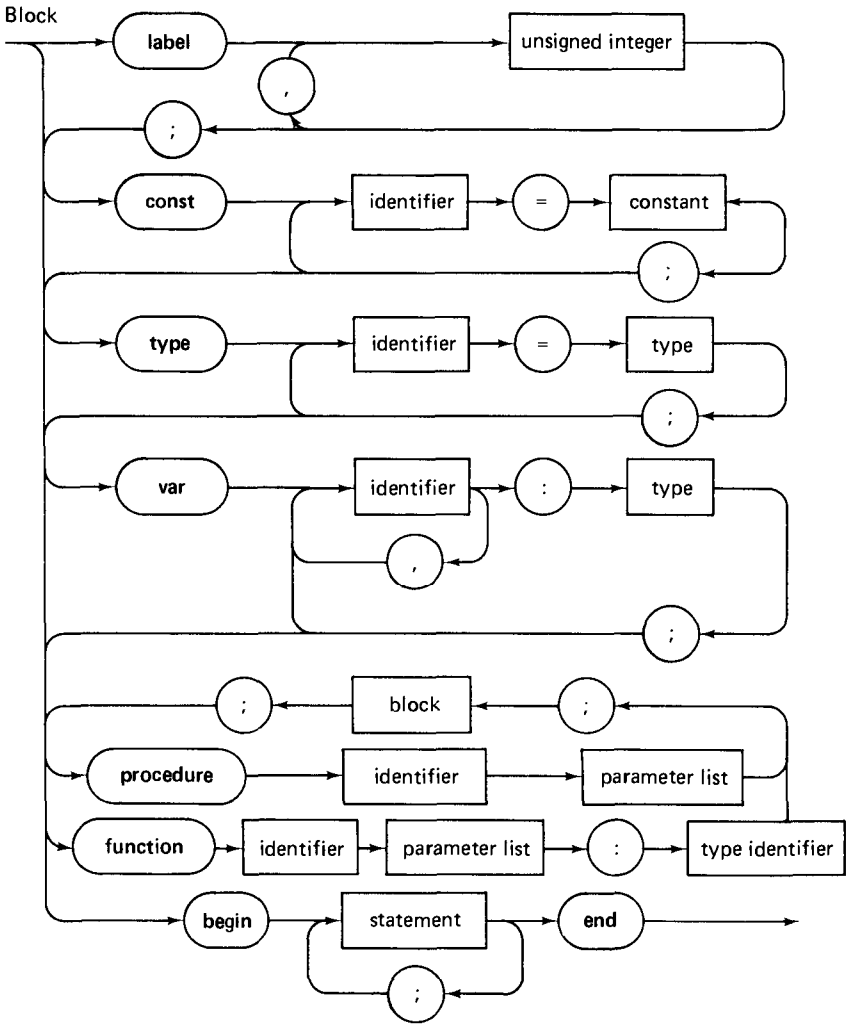
Type

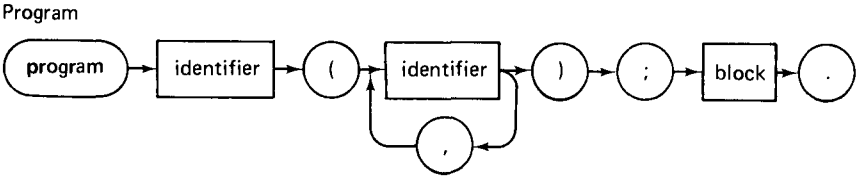


Field list




Statement
unsigned integer
:
variable
:=
expression
function identifier
,
procedure identifier
(
expression
)
procedure identifier
begin
statement
end
;
if
expression
then
statement
else
statement
case
expression
of
constant
:
statement
end
,
;
while
expression
do
statement
repeat
statement
until
expression
;
downto
for
variable identifier
:=
expression
to
expression
do
statement
,
with
variable
do
statement
goto
unsigned integer


Block
label
unsigned integer
,
;
const
identifier
=
constant
;
type
identifier
=
type
;
var
identifier
:
type
,
;
;
block
;
procedure
identifier
parameter list
function
identifier
parameter list
:
type identifier
begin
statement
end
;


Program
program
identifier
(
identifier
)
;
block
.
,