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.

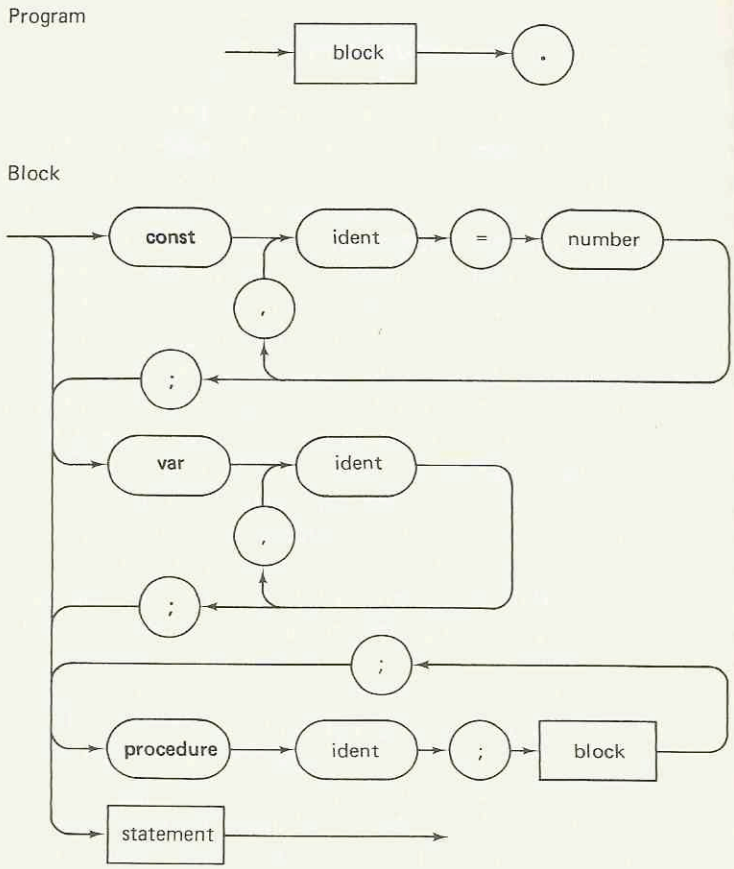
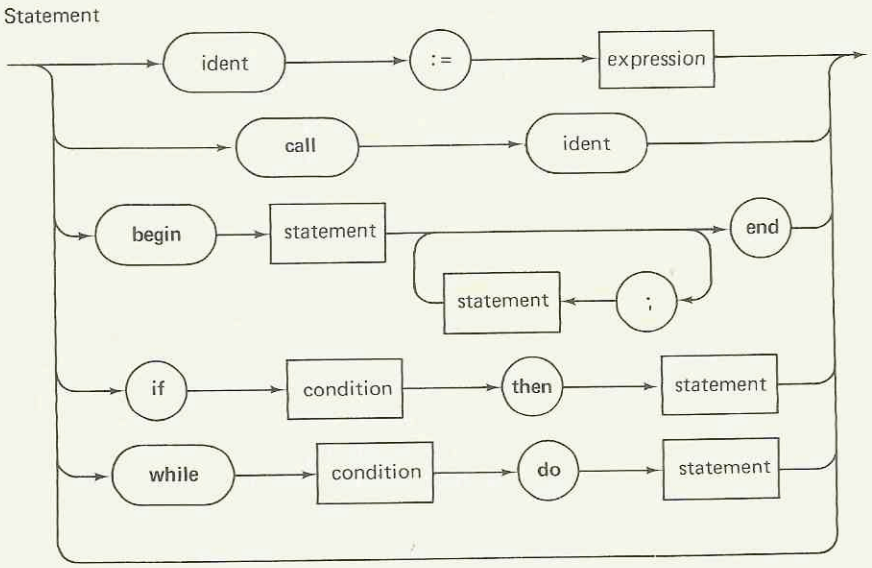


Fig. 5.4 Syntax of PL/0.



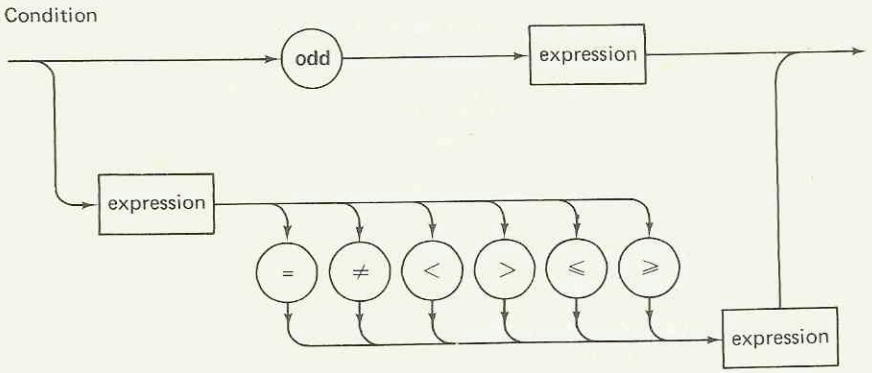


Fig. 5.4 (Continued)

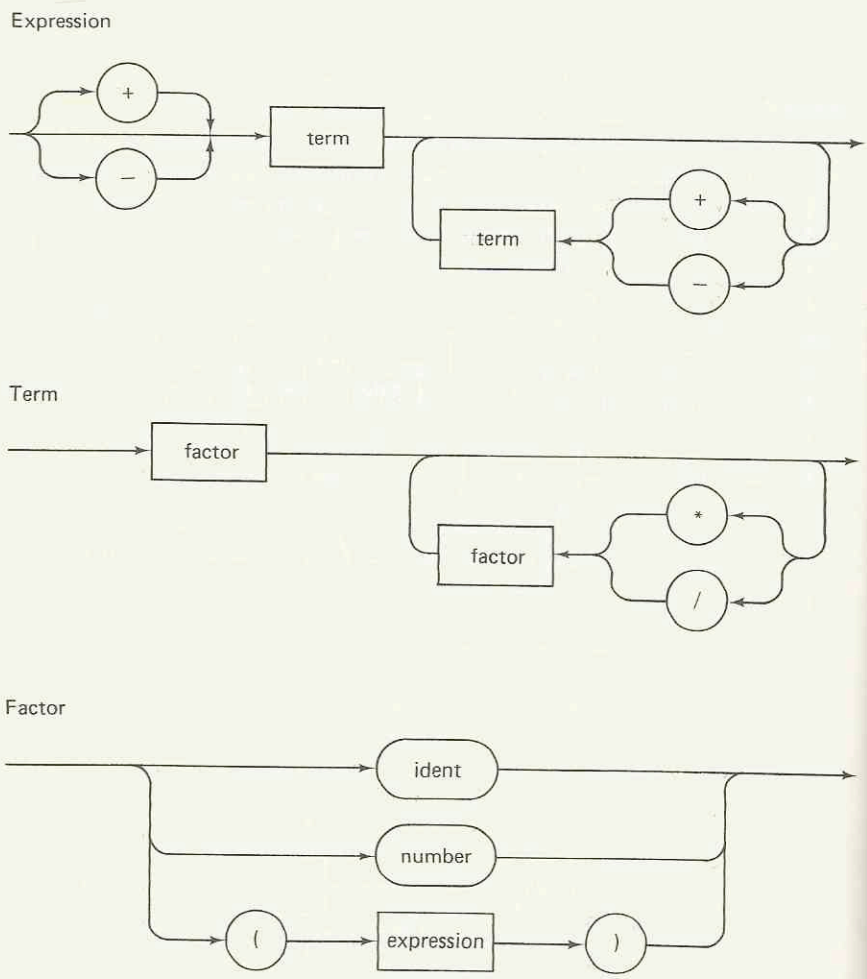


Fig. 5.4 (Continued)

```
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
            if odd b then z := z + a;
            a := 2*a; b := b/2;
        end
end ;
```

(5.14)

```
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.

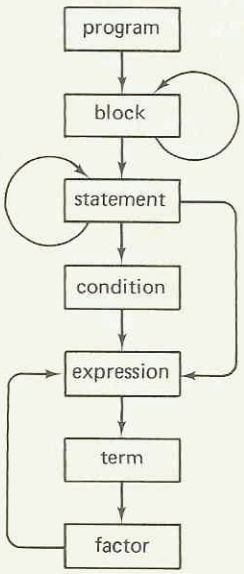
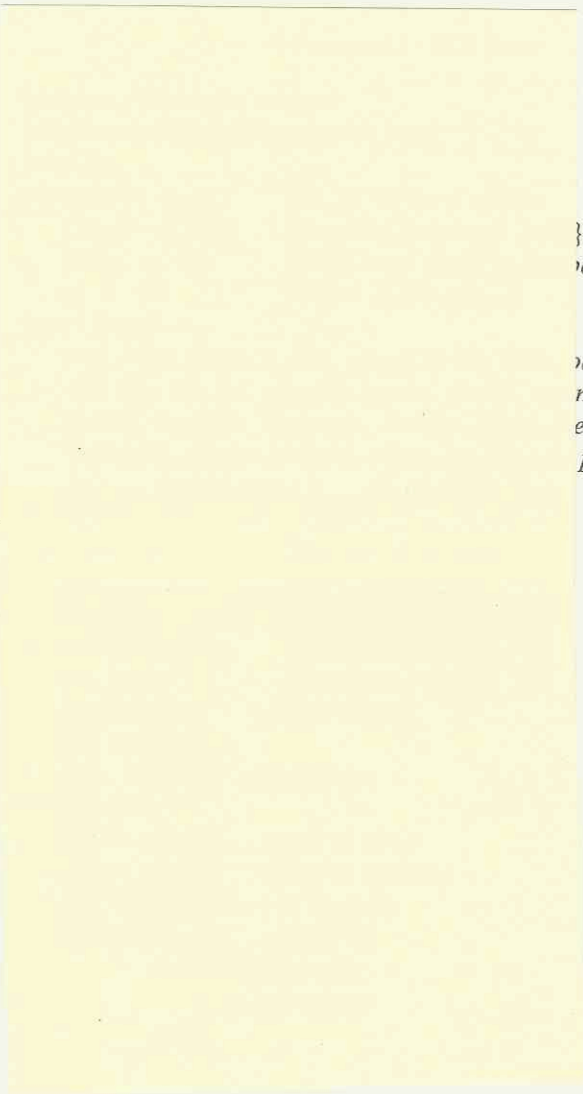


Fig. 5.5 Dependence diagram for PL/0.

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*.



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

$$A ::= a_1 S_1 | \ldots | a_n S_n | X \tag{5.18}$$

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

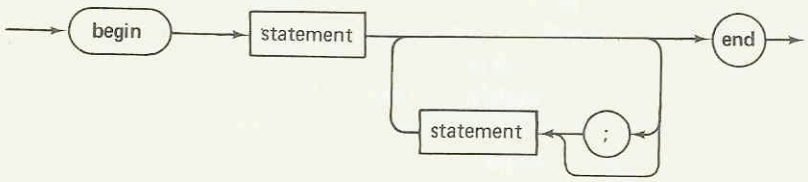


Fig. 5.6 Modified compound statement syntax.

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.

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*!

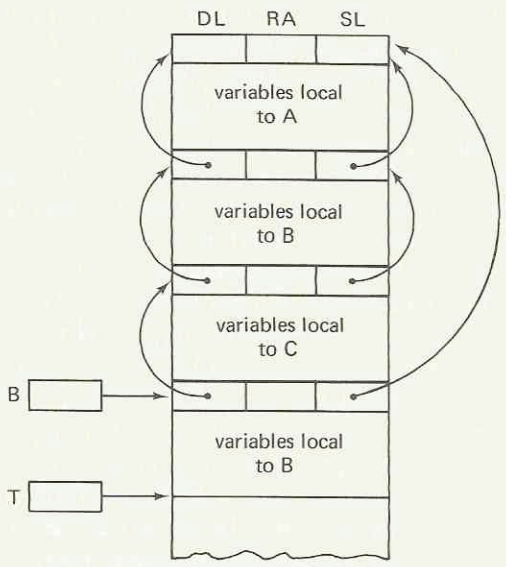


Fig. 5.7 Stack of PL/0 machine.

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).

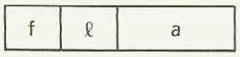


Fig. 5.8 Instruction format.

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].