Eidgenössische
Technische
Hochschule
Zürich

*Institut
für
Informatik*

Niklaus Wirth

PASCAL-S: A Subset and its Implementation

Juni 1975

12

Eidgenössische
Technische
Hochschule
Zürich

*Institut
für
Informatik*

Niklaus Wirth

PASCAL-S: A Subset and its Implementation

PASCAL-S: A Subset and its Implementation

N. Wirth

Abstract. Pascal-S is a subset of the programming language Pascal selected for introductory programming courses. This report describes an implementation that is especially designed to provide comprehensive and transparent error diagnostics and economical service for large numbers of small jobs. The system consists of a compiler and an interpreter and is defined as a single, self-contained Pascal program. This machine-independent formulation in a high-level language facilitates its construction and is a prerequisite for easy portability.

Institut fuer Informatik ETH
Clausiusstrasse 55
CH-8006 Zuerich

<u>Contents</u>

1. Aims and Motivation

Several years ago, the Computer Science Department of ETH Zuerich had started to use the programming language Pascal in its introductory programming courses [2,3]. These courses are taught mainly to engineers, physicists, and mathematicians in their first year. The large number of participants dictates the use of an efficient and economical system; economical with regard to the students' learning effort, to computing time, and to storage requirements. The first demand requires a system with comprehensive syntax and run-time error checking and the provision of meaningful, well-explained diagnostics. Machine economy was realised through a combination of a compiler and a sub-batch monitor, the latter alternately invoking compilation and program execution. This scheme requires an absolutely watertight protection against errors in compiled programs and--of course--an entirely errorfree compiler and monitor.

The system developed for this purpose by R. Schild proved to be highly successful, and it turned out to be so economical with respect to computing time that the system's extensive use by about 350 students amounted to less than 0,5 % of the entire computing services provided by the computation center averaged over the entire term, although the student job batch was collected, run, and returned four times per day.

Nevertheless, there were also some disadvantages. First of all, there were only four fixed times when jobs were collected (and sometimes fewer due to machine failures). Then, the student jobs had to be handled separately from all other jobs (separate batching and special job cards to be provided by the operators). These drawbacks and the advent of a separate medium speed batch terminal located in the students program preparation room indicated that a system with different characteristics might be more appropriate. The students' batch terminal allows for self-service. It is therefore highly desirable - if not mandatory - that each student's job be scheduled and run independently of other jobs. Fast turn around can - under the prospect of large numbers of jobs - only be guaranteed, if the jobs use relatively little store. In fact, storage space is at a much higher premium than processor time.

Under these conditions, a compact system is mandatory. Without compromising on the demand for extensive error checking, our interpretive solution appears as most promising, because it allows for a simple compiler and dense code. Program size can be further reduced, if the system is restricted to handling that subset of Pascal, which is actually taught in these introductory courses. Hence, the new system was intentionally designed to process a subset. The resulting reduction of development labor was an additional incentive for this decision.

In choosing an interpretive approach, one must be aware of and willing to accept a substantial loss of efficiency in program execution. A factor of 30 compared to reasonably good compiled code is not unusual, and a factor of 20 must be called "very good". Such factors can of course only be accepted, if the gains expected elsewhere are equally substantial. They can only be compensated, if the execution effort of the compiled program is relatively small, say at least 20 times smaller than job initiation, compiler loading, compilation, and program loading together.

In view of the everyday performance figures of common operating systems, this condition is indeed satisfied for very many problems that can successfully be assigned as programming exercises. The most important single factor is gained by eliminating the need for a relocation loader. This is achieved by directly depositing the compiled code in store. As programs tend to be small, even the demand for storage economy cannot be used as a strong counterargument against this strategy.

The system described in this report consists of a compiler and an interpreter for a subset of Pascal called Pascal-S. Chapter 2 defines that subset; it contains a complete syntax specification in terms of concise syntax diagrams. These diagrams directly mirror the structure of the compiler. Chapter 3 provides an overview of the entire system which is described as a single Pascal program. Some figures are provided concerning system size and performance. Chapter 4 explains the architecture of the hypothetical computer that executes compiled Pascal-S programs. Some typical program constructs are listed together with the code generated for them. The principles of operation of the computer become understandable through these sample constructs which at the same time picture the task of translating Pascal-S into this code. Chapter 5 discusses the compiler itself, and it starts out with an explanation of the tables and their structure used to represent the information given in a program's declarations.

The explanations are necessarily terse and brief and incomplete. They are intended for people who already have some background on compilers; they are in particular referred to [3], where the principles along which this system is constructed are taught and developed. For all details, the reader is referred to Chapter 6 which is a full listing of the entire system. One may wonder about the value of including a program listing in extenso. But I think it is important and hope it will be useful. The primary value of the language and system Pascal is that it allows to construct large programs that are useful and highly efficient in a form that can <u>be</u> read and communicated. The listing of the Pascal-S system is intended to support this claim. It also proves that compiler <u>and</u> interpreter can be described in a machine-independent, well-structured form that nevertheless is effectively machine translatable. The relative brevity of the program (25 pages) also raises a new aspect of compiler portability: it is entirely possible to transport such a system by hand coding. The effort is at most one of a few man months, even for a computer where nothing but symbolic assembly code or Fortran are available.

2. The language PASCAL-S

The choice of features to be included in the subset now called Pascal-S was mainly guided by the contents of traditional introductory programming courses. Beyond this it is subject to personal experience, judgement, and prejudice. A firm guideline was provided by the demand that the system must process a strict subset of Pascal, i.e. that every Pascal-S program must also be acceptable by the compiler of Standard Pascal without being subjected to the slightest change. This rule makes it possible for students to switch over to the regular system in later courses "without noticing". A languages's power and its range of applications largely depend on its data types and associated operators. They also determine the amount of effort required to master a language. Pascal-S adheres in this respect largely to the tradition of Algol 60. Its primitive data types are the integers, the real numbers, and the Boolean truth values. They are augmented in a most important and crucial way by the type char, representing the available set of printable characters. Omitted from Pascal are the scalar types and subrange types.

Pascal-S includes only two kinds of data structures: the array and the record (without variants). Omitted are the set and the file structure. The exception are the two standard textfiles input and output which are declared implicitly (but must be listed in the program heading). A very essential omission is the absence of pointer types and thereby of all dynamic structures. Of course, also all packing options (packed records, packed arrays) are omitted.

The choice of data types and structures essentially determines the complexity of a processing system. Statement and control structures contribute but little to it. Hence, Pascal-S includes most of Pascal's statement structures (compound, conditional, selective, and repetitive statements). The only omissions are the with and the goto statement. The latter was omitted very deliberately because of the principal use of Pascal-S in teaching the systematic design of well-structured programs. Procedures and functions are included in their full generality. The only exception is that procedures and functions cannot be used as parameters.

The detailed syntax of Pascal-S can be seen from the syntax diagrams which are included in the Appendix. They reveal a simple and consistent language that can be learned in toto in a very short time, yet encompasses many of the truly fundamental concepts of programming.

Teaching experience over many years has shown that the concept of the sequence (sequential file) is of fundamental importance for the understanding of many computing practices and techniques. Inspite of the absence of declarable files, it can be taught quite well with Pascal-S because of the presence of the two standard textfiles. We have deliberately decided to exclude the primitive operators put and get, and have restricted file operations to <u>read</u> (on input) and <u>write</u> (on output). Also included are the simple but flexible "formatting" facilities of the Pascal write-statement. They proved to be not only useful and desirable from the point of view of utility, but indeed also quite simple to teach. (Actually, they don't even need to be taught; students learn to use them quite naturally, as they are entirely free of pitfalls.)

The standard objects available in Pascal-S are:

Constants:	true, false
Types:	integer, real, Boolean, char
Functions:	abs, sqr, odd,
	chr, ord, succ, pred,
	round, trunc,
	sin, cos, exp, ln, sqrt, arctan,
	eof, eoln,
Procedures:	read, readln, write, writeln

Functions pred and succ are only applicable to arguments of type char. The argument of ord can be of type char, Boolean, or integer. For further details concerning the language the reader is referred to the literature [2,4]. It is noteworthy that the subset corresponds largely to that part of the language Pascal which is covered in a textbook for an introductory programming course [4,5].

3. <u>The Implementation</u>

The Pascal-S system is described as a compiler that translates Pascal-S programs into code for a hypothetical stack computer especially designed for this purpose. This computer is itself defined as an algorithm, called the <u>interpreter</u> of the compiled code. Both compiler and interpreter are described in a largely machine-independent way by using the high-level language Pascal exclusively. In fact, these two parts form a single Pascal program. It is listed in Chapter 6.

The advantages of a description using a high-level language are particularly apparent during the development of a system, but are equally significant, if it has to be transported and adapted to a different computer. In fact, Pascal-S can be implemented immediately on all machines where a full Pascal compiler is available. Of course, the success of such an automatically generated system crucially depends on the quality of the tool compiler. The Pascal 6000-3.4 compiler used at ETH on the CDC 6400 computer generates high-quality code, and recompiles the entire Pascal-S system in 20 sec.

The disadvantage of an interpretive system is its relatively large overhead during program execution. Experience has shown, however, that for small programs the expense for central processor utilization for interpretation is anyway quite small. Some actual figures are given for a few sample programs in Chapter 7. Exercises requiring less than 4 seconds of computer time (costing less than SFr. 2.-) are predominant. Of course, exercises will have to be carefully chosen, particularly in numerical mathematics.

Apart from its machine-independent specification, the following characteristics are noteworthy.

1. The system resides on the disk store as an absolute binary overlay file; hence, loading is fast.
2. The storage space needed is reasonably small. On the CDC 6000 computer it requires 10'000 60-bit words, including data and I/O buffers.
3. The compiler is designed to recover from syntax errors and to proceed after emitting a diagnostic keyword. This policy usually allows many errors to be detected from a single compilation. A significant effort is made to suppress so-called "spurious" error messages, i.e. indication of irregular situations that are due to previously reported errors.
4. Control is not returned to the operating system between compilation and execution. No loader is invoked nor is secondary storage accessed to deposit the generated code. This resulted in a very significant saving of overhead and cost.
5. A copy of the input data is made on the output file immediately after compilation. This is often an invaluable aid to consultants and tutors.
6. The interpretation steps needed for program execution are counted, and provide a precise, reproducible hardware-independent measure for the actual computational effort expended.
7. By exchanging a single operating system control card the student may switch over to the regular, full Pascal compiler. Pascal-S is a true subset of Standard Pascal.
8. If an error is detected at run-time, execution is aborted and a listing of the names and values of all currently accessible variables is printed, together with the coordinate of the point of interruption and an indication of the reason for the interruption (post mortem dump).
9. The system requires no access to secondary store except for the standard input and output files. The amount of these data is usually so small that a single access (blocktransfer) is sufficient for each file.

4. The Interpreter

The Pascal-S system consists of two main parts: compiler and interpreter. Their principal interface is the array variable to which the compiler assigns the generated code. The interpreter itself is formulated as a procedure which is called after successful compilation.

The interpreter describes a straight-forward stack computer, consisting of a store S organised as a stack two index registers T and B which control the stack, program counter PC , an instruction register IR , a program status register PS and a DISPLAY used to speed up the addressing mechanism. Each element of the stack represents either an integer, a real number, a logical value, or a character. The principal structure is

```
   procedure interpret;
      begin initialise registers and auxiliary counters;
           repeat IR := code[PC]; PC := PC+1;
                  interpret(IR)
           until PS ≠ run;
           if PS ≠ fin then postmortemdump
      end
```

Each instruction (order) is characterised by an order code f with values between Ø and 63. Orders with values Ø, 1, 2, 3 have two parameters x and y . Instructions Ø, 1, 2 generate an address of the data element on the stack with offset y in the currently active data segment on level x . Orders with codes 8 through 3Ø have a single parameter y whose meaning differs in the indiviual cases. Orders with codes 31...63 have no parameters and are operators whose argumets are the elements on the top of the stack.

We refrain from introducing a complete set of mnemonics for the orders. Instead, short key words are given as comments in the listing of the interpreter where necessary. The individual routines are in most cases simple enough to make any commentary superfluous. We can therefore restrict ourselves to a presentation of the general layout of the stack and of the patterns of emitted code for specific language constructs. These will show where individual orders are used, and thereby make their principles of operation understandable.

4.1 Storage layout and procedure calls

Each stack element may either be an integer, a real number, a logical value, or a character. Integers are also used as stack indices. Each activated procedure reserves a stack section on the top. The beginning of each section is designated by a mark which contains a pointer to the previous section. These pointers form the so-called dynamic link, as they record the dynamic history of procedure activations. There is also a static link, which connects those sections that belong to procedures declared within each other. This chain designates all segments which are to be currently accessible. The static chain starting with the currently active procedure is also copied into the short array called display. This is done to speed up data access.

The first five locations of each stack section are occupied by the so-called section mark. The mark contains the two links, the return address, a pointer (index) to the symbol table, and the result value (used by function procedures only). All data are accessed by an offset address relative to the section origin, or via the top stack pointer. Subsequent locations are used for procedure parameters, followed by local variables. The top of the stack is used for intermediate results in evaluating expressions.

Stack sections are "allocated" when procedures are activated by changing the stack pointers and setting up the two links. First, the stack is "marked", i.e. a space for the block mark is reserved. Then the actual parameters are processed. In the case of value parameters the actual values are loaded onto the stack; in the case of variable parameters, stack addresses are loaded instead of values. Finally, the procedure call order changes the B and T registers and assigns the links and return address to the section mark.

Upon procedure exit, the return order reverses the operations performed by the call order. If the static level of the current procedure is lower than the one of the procedure to which control return, then the display has to be updated. This is done by a separate order which indicates the two levels.

4.2 Control structures

Control structures are translated into sequences of instructions containing jump orders according to the following patterns.

```
1)  if B then S
2)  if B then S1 else S2
3)  while B do S
4)  repeat S until B
5)  for i := A to B do S
6)  for i := A downto B do S
7)  case E of
          v1: S1; v2: S2; ... ; vn: Sn
    end
```

```
1)     code(B)                      2)     code(B)
       conditional jump to L               conditional jump to L1
       code(S)                             code(S1)
   L:  ...                                 jump to L2
                                       L1: code(S2)
                                       L2: ...

3) L1: code(B)                      4) L:  code(S)
       conditional jump to L2              code(B)
       code(S)                             conditional jump to L
       jump to L1
   L2: ...

5)     load address i               6)     load adress i
       code(A)                             code(A)
       code(B)                             code(B)
       for1up L2                           for1down L2
   L1: code(S)                         L1: code(S)
       for2up L1                           for2down L1
   L2: ...                             L2: ...

7)     code(E)
       search switchlist L
   L1: code(S1)
       jump to K
   L2: code(S2)
       jump to K
       ...
   Ln: code(Sn)
       jump to K
   L:  (v1,L1)
       (v2,L2)
       ...
       (vn,Ln)
   K:  ...
```

In the case of the for statement, the address of the control variable and the two limit values are left on the stack during execution of the repeated statement. The switch order used in the case statement performs a simple, linear search through the switch list, comparing the value on top of the stack with entries v1...vn . If a match vi is found, a jump to Li is executed.

4.3 Post mortem dump

If interpretation of the code leads to an error condition, execution is terminated and a symbolic post mortem dump is generated. Detected error conditions are:
- division by Ø
- selector value of a case statement is out of range
- array index out of bounds

- stack overflow
- line limit exceeded (too many lines)
- output line too long
- attempt to read beyond the end of the input file.

The post mortem dump consists of a list of the currently active procedures with indication of their activation points, and (for each procedure) a list of its local variables and their current values. In order to keep the amount of information reasonably small, only unstructured variables are listed, as they usually contain the information relevant for detecting the cause of a trap. An example of a dump follows:

```
    0  program runerror(output);
    0    var i: integer; b: boolean; x: real;
    0
    0    function f(m,n: integer): integer;
    0    begin f := f(n, m mod n) end ;
    9
    9  begin x := 9.87654321; b := true; i := f(511,31)
   20  end .
 (eof)
```

```
halt at    6 because of division by 0
 f           called at     7
   n             =         0
   m             =         1
 f           called at     7
   n             =         1
   m             =        15
 f           called at     7
   n             =        15
   m             =        31
 f           called at    21
   n             =        31
   m             =       511
   x             =    9.876543210000E+000
   b             =      true
   i             =      8506
        38 steps
```

5. The Compiler

Pascal is a language that can be parsed with a lookahead of a single symbol. The compiler therefore uses the simple and efficient method of top-down parsing with one-symbol lookahead. It is organised as a set of procedures, each representing a specific sentential construct and parsing goal. These procedures may activate each other recursively, just as certain sentential constructs occur recursively. The parsers obtain their input through a scanner called insymbol. This scanner reads the input file character by character, and delivers the next Pascal symbol each time it is called. For this purpose, the scanner requires a lookahead of one character. The total lookahead of the compiler is therefore one symbol plus one character. For further information on the principles of operation of a top-down, recursive descent compiler the reader is referred to [6].

When working his way through the compiler listing, the reader is advised to start with the scanner called Insymbol. The next symbol read is assigned to the global variable sy (which represents the symbol lookahead). The scanner receives its input by calling procedure nextch , which assigns the next character read to the global variable ch (which represents the character lookahead). If an identifier is encountered, the actual identifier is assigned to the global variable id , and if it is a number, its value is assigned to the global variables inum or rnum . If the symbol is a string, the string is directly assigned to the string table stab (note that strings may occur only as parameters in write statements).

The set of procedures used to parse and translate Pascal-S programs closely mirrors the syntactic structure of the language. The reader is advised to consult the syntax diagrams, as they represent abstract flow-charts of the parser procedures. It is useful to keep the following compiler excerpt in mind which mirrors the way in which the compiler is partitioned and exhibits the interdependence of the principal procedures (see also the Procedure Dependence Diagram in the Appendix).

```
block
   constant
   typ
      arraytyp
   parameterlist
   constantdeclaration
   typedeclaration
   variabledeclaration
   proceduredeclaration
   statement
      selector
      call
      expression
         simpleexpression
            term
               factor
               standard functions
      assignment
      compound statement
      if statement
      case statement
      repeat statement
      while statement
      for statement
      standard procedures
```

Much effort is spent at providing robustness against ill-formed input recovery and at obtaining sensible error diagnostics. To achieve this aim, a systematic approach to syntax error handling is used [1,6]. Its main principle is that each parser always returns control after having advanced up to a symbol that may legally follow the sentential construct that the parser is supposed to process. If the input program contains errors, this goal is usually achieved by skipping text until such an acceptable symbol is encountered. For this purpose, procedures skip and test are used.

The scheme requires that each parser know the set of symbols that may legally follow its sentential construct in the current context. To this aim, each parser is provided with a parameter indicating that set of so-called follow symbols (called fsys). This set is, however, augmented by certain key symbols which are never to be ignored. Typically, these key symbols are those which head a specific sentential construct, such as begin, if, type, etc. Hence, these parameters do not necessarily specify legal follow symbol, but rather the symbols where a possible skip has to terminate.

The following test program shows the compiler's handling of syntactically ill-formed texts. Adequate recovery from syntactic errors is indeed a crucical criterion for a system to be used in an environment where errors occur frequently. A list of brief explanations to the error numbers is included in the Appendix.

```
   0  program syntaxerror(output);
   0    const m = 10, n := 20
****                    ↑14  ↑16
   0    type t = array 1..10 of real;
****       ↑14            ↑11   ↑12
   0      r := record x: real,
****        ↑16                ↑14
   0            b,c : boolean
   0          end
   0    var i: integer;
****      ↑14
   0        p,q: boolean; x,y: real;;
****                                ↑ 6
   0        i: integer, ch: char
****        ↑ 1        ↑14
   0        a: array (1..m) of integer;
****        ↑14      ↑11  ↑12
   0    const y = 3.14159;
****        ↑56
   0  begin i := x   m := i
****                 ↑ 6
   3    if b do p = (p or q:
****       ↑14 ↑52 ↑51      ↑ 4
  12    while j < 10
****          ↑ 0
```

```
  14      begin k := .5+(x-y;  y := x)
****          ↑35  ↑14             ↑ 6
  18      end
  19    if p then p = 1; else i := 2;
****     ↑14        ↑51      ↑14
  24    repeat x := p + i*(x>y);
****                          ↑33
  30      for x = 1 to q
****            ↑18
  31        begin i := a[2
****            ↑19
  36    until i=j
****        ↑28 ↑ 0
  39    for j := 1 to n while x > 0 do
****      ↑35            ↑54
  50      begin a(j] := a[j+1); read(i)
****             ↑11      ↑ 0 ↑28   ↑20
  59  end .
****      ↑14
program incomplete
key words
        0  undef id
        1  multi def
        4  )
        6  syntax
       11  [
       12  ]
       14  ;
       16  =
       18  convar typ
       19  type
       20  prog.param
       28  no array
       33  arith type
       35  types
       51  :=
       52  then
       54  do
       56  begin
```

The compiler can functionally be subdivided into two main parts: the part processing declarations, and the part processing statements and expressions. Their common interface is the symbol-table tab and further associated tables. They are constructed by the declaration processing part, and constitute the necessary context in which the program statements are to be compiled. Knowledge of the structure of these tables is therefore of fundamental importance. The key table is tab. Each declared identifier causes one entry. All entries of identifiers local to the same procedure (block) are linked together. Note that the compiler program is written without use of any dynamic structures and of pointers. Hence, a linked chain is represented by explicit array indices. The field called obj indicates whether the identifier denotes a constant, a variable, a type, a procedure, or a function. Its type is specified by the field called typ. The meaning of the remaining fields varies according to object and type. If the type of an entry is an array type, then the ref field is an index to the table of array structures called aref; if it is a record type, the ref field contains an index to the table of records and blocks called btab. The Boolean field normal specifies whether an entry is an actual (normal) variable to be addressed directly or a formal parameter to be addressed indirectly. The fields lev and adr specify the address pair of a variable or procedure. If the entry denotes a constant and the constant is of type integer, Boolean, or char, then the adr field indicates its value. If its type is real, the adr field specifies an index of the table of real numbers called rconst.

The array table atab specifies for each array structure its index type and index bounds, its element type (eltyp and elref, where the latter is used in analogy to the field ref above), and its size in terms of storage elements. For the sake of convenience only, the size of an element is also present, although it could easily be derived via eltyp and elref.

Each procedure and each record type definition causes an entry in the table of "blocks" called btab. It contains pointers to the last identifier (in tab) defined local to the block and to the last parameter of the corresponding procedure. Note that all previous entities can be accessed through the linked chain. Moreover, the entry specifies the storage size needed to represent the set of variables belonging to the respective record or procedure.

A sample program and the structure of the tables constructed during its compilation is shown below. These tables are derived from the auxiliary output generated by the compiler itself. Note that these tables are not released (collapsed) after exit from a block, as the information gathered may be required to generate a post mortem dump. They are also accessed by some orders during program execution, e.g. by procedure calls and index orders. This makes it unnecessary to copy certain information into the code (e.g. array index bounds and data segments lengths), thereby contributing to code density.

```
0  program test0(output);
0    const ten = 10; plus = '+';
0    type row = array [1..ten] of real;
0           complex = record re,im: real end ;
0    var i,j: integer;
0          p: boolean;
0          z: complex;
0          matrix: array [-3..+3] of row;
0          pattern: array [1..5, 1..5] of char;
0
```

```
 0     procedure dummy(var i: integer; var z: complex);
 0       var u,v: row;
 0         h1,h2: record c: complex; r: row
 0                end ;
 0
 0       function null(x,y: real; z: complex): boolean;
 0         var a: array ['a'..'z'] of complex;
 0             u: char;
 0       begin while x < y do x := x+1.0;
10             null := x=y
12       end (*null*) ;
15
15     begin p := null(h1.c.re, h2.c.im, z)
25     end (*dummy*) ;
27
27   begin i := 85; j := 51;
34     repeat
34       if i > j then i := i-j else j := j-i
46     until i = j;
53     writeln(i)
55   end .
```

```
identifiers                link   obj   typ   ref   nrm   lev   adr
         29 ten               0     0     1     0     0     1    10
         30 plus             29     0     4     0     0     1    37
         31 row              30     2     5     1     0     1    10
         32 complex          31     2     6     3     0     1     2
         33 re                0     1     2     0     1     2     0
         34 im               33     1     2     0     1     2     1
         35 i                32     1     1     0     1     1     5
         36 j                35     1     1     0     1     1     6
         37 p                36     1     3     0     1     1     7
         38 z                37     1     6     3     1     1     8
         39 matrix           38     1     5     2     1     1    10
         40 pattern          39     1     5     3     1     1    80
         41 dummy            40     3     0     4     1     1    16
         42 i                 0     1     1     0     0     2     5
         43 z                42     1     6     3     0     2     6
         44 u                43     1     5     1     1     2     7
         45 v                44     1     5     1     1     2    17
         46 h1               45     1     6     5     1     2    27
         47 h2               46     1     6     5     1     2    39
         48 c                 0     1     6     3     1     3     0
         49 r                48     1     5     1     1     3     2
         50 null             47     4     3     6     1     2     0
         51 x                 0     1     2     0     1     3     5
         52 y                51     1     2     0     1     3     6
         53 z                52     1     6     3     1     3     7
         54 a                53     1     5     5     1     3     9
         55 u                54     1     4     0     1     3    61

blocks       last  lpar  psze  vsze
          1    28     1     0     0
          2   41   28    5  105
          3   34    0    0    2
          4   50   43    7   51
          5   49    0    0   12
          6   55   53    9   62

arrays      xtyp etyp eref  low high elsz size
          1    1    2    0    1   10    1   10
          2    1    5    1   -3    3   10   70
          3    1    5    4    1    5    5   25
          4    1    4    0    1    5    1    5
          5    4    6    3    1   26    2   52

code:
 0     1 3     5,    1 3     6,   41         ,   11       10,    0 3     5,
 5     1 3     5,   25       1,   54         ,   38         ,   10        0,
10     0 3     0,    1 3     5,    1 3     6,   39         ,   38         ,
15    33         ,    0 1    7,   18       50,    0 2     27,   34         ,
20     0 2    39,    9       1,   34         ,    1 2      6,   22        2,
25    19       8,   38        ,   32         ,    0 1      5,   24       85,
30    38         ,    0 1    6,   24       51,   38         ,    1 1      5,
35     1 1     6,   49        ,   11       44,    0 1      5,    1 1      5,
40     1 1     6,   53        ,   38         ,   10       49,    0 1      6,
45     1 1     6,    1 1     5,   53         ,   38         ,    1 1      5,
50     1 1     6,   45        ,   11       34,    1 1      5,   29        1,
55    63         ,   31        ,
(eof)
      17
      51  steps
```

6. Machine-dependencies

Every program must be tailored to the facilities that are available in the language and the computing system used. It is desirable to restrict these considerations to the high-level language (in this case Pascal) and to ignore the actual hardware. In the case of a program to be used thousands of times and every day by many people, this is only possible and economically justifyable, if the language successfully hides the hardware without causing appreciable loss in efficiency. Although the Pascal 6000-3.4 compiler satisfies this requirement to a high degree. We nevertheless had to resort to facilities that are not available in Standard Pascal in a few instances. These facilities are particular to the Pascal implementation on the CDC computer and are listed in detail below. Their use introduces what may be called first-order machine dependencies.

The only such facility used is the Segmented File. It allows to recognise a substructure of the file called segment (in Pascal terminology) or "logical record" (in CDC terminology. The fact that the file input is to be treated like a segmented file is indicated by a plus sign in the program parameter list (implemented through Update 10 of Pascal 6000-3.4). A job is represented in the CDC operating system as an input file consisting of three segments: control statements for the operating system, program, and data. The recognition of this substructure is essential for Pascal-S in order to skip backward to list the input data segment, and it is desirable in order to keep the rules for setting up a Pascal-S job deck identical to those for all other jobs, particularly normal Pascal jobs. The reader is referred to [2], Manual section 13.A.1 for an explanation of the procedure getseg and the predicate eos.

But even when strictly adhering to a machine-independent language such as Standard Pascal, a second kind of machine-dependent considerations creep in, if a program is carefully planned. I shall call them second-order machine dependencies. They are due to the use of knowledge about limitations and characteristics of the underlying system and the desire to use it optimally. They may cause another implementation to reject the program (if its limitations are more severe) or merely to process the program less efficiently. The first category concerns for instance the range of available integers or - much more problematic in Pascal - the size of allowable sets. The second category includes considerations of storage structure. In the present system program, such considerations played an important role in achieving high efficiency and economy, and are manifested in the choice of several constants which are all defined in the beginning of the program . The choice of these constants, explained below, must be reconsidered if reimplementing Pascal-S on a different computer, be it by hand translation or by recompilation through an already available Pascal compiler. (The reader not concerned with this problem may easily skip the rest of this chapter.)

alng	defines the number of characters in the array type alfa In a word-oriented computer the choice of this value is critical, and should be the number of characters packable into a word or a small number of words.
llng	defines the maximum length of an input line delivered by the operating system.
emax	this is the maximum value of the decimal exponent of a real number acceptable by the computer emax = log(maxreal)
emin	is the minimum value of the decimal exponent. (Smaller numbers are considered as identical to 0.) (For most computers emin = -emax.)
kmax	is the number of significant digits in a real number, i.e. kmax = m log 2, if m is the number of bits of the mantissa of the binary floating-point number.
ermax	is the number of error messages available. It is chosen such that the type errs was acceptable to the available Pascal compiler.

omax, lmax, nmax	were chosen such that the packed record type *order* could be represented in a single 60-bit word of the CDC computer.
xmax	is chosen such that the subrange type index occupies a reasonably small part of a word to provide high storage economy in packed tables, yet encompasses a sufficiently large subset of the integers to cover all array index values.
lineleng	is equal to the maximum number of characters permissible in a line to be printed.

Further comments about second-order machine dependencies follow. The type *order* allows for negative values of components f and x, although this is not needed. The reason is that access to signed fields of packed records is more efficient in the Pascal 6000-3.4 system, and because these accesses are very frequent in the interpreter.

The number of basic symbols must be such that the type *symset* is acceptable to the available Pascal compiler. The array *sps* is a constant table used by the scanner. Its index range must be such that it covers all characters which are neither letters nor digits. For the sake of character set independence, its range is indicated as being the entire character set.

The set constants used in the scanner depend on the assertion that

```
     ch in ['a'..'z']  =  ch is a letter
     ch in ['0'..'9']  =  ch is a digit
```

Most available character sets satisfy these equivalences. Several statements in the scanner rely on them by computing the numeric value of a digit x as ord(x)-ord('0'). This is the only requirement imposed on the ordering of character sets. The present version of Pascal-S assumes the use of the restricted ASCII character set. Unused characters cause an error indication when encountered.

In the interpreter, the functions *chr* and *ord* are implemented as dummy procedures, because we postulate for characters the ordering given by the collating sequence of the given character set (thereby accepting an implicit machine dependence). The constants 0 and 63 are the ordinal numbers of the first and the last character in the given set.

Note that an element in the stack may represent a value of any of the four standard types integer, real, Boolean, or char. This implies that the same amount of storage is allocated for values of these types. This results in uneconomical utilization of store, particularly for Boolean and character values, but it simplifies both compiler and interpreter considerably. With regard to real numbers, a floating-point representation should be chosen that uses the same number of words or bytes as integers, since a very high numerical precision is usually not required in the types of problems for which Pascal-S is intended.

A particular problem of practical importance is the regaining of control in the case of a trap performed by the hardware or the underlying operating system. No solution for this problem is indicated here, as it is inherently dependent on the environment.

7. The compiler-interpreter program

```
program Pascals(input+,output);       (*1.6.75*)
(*        N. Wirth,  E.T.H.
          Clausiusstr.55   CH-8ØØ6 Zurich    *)
label 99;
const nkw = 27;        (*no. of key words*)
    alng =  1Ø;        (*no. of significant chars in identifiers*)
    llng = 12Ø;        (*input line length*)
    emax = 322;        (*max exponent of real numbers*)
    emin =-292;        (*min exponent*)
    kmax =  15;        (*max no. of significant digits*)
    tmax = 1ØØ;        (*size of table*)
    bmax =  2Ø;        (*size of block-table*)
    amax =  3Ø;        (*size of array-table*)
    c2max = 2Ø;        (*size of real constant table*)
    csmax = 3Ø;        (*max no. of cases*)
    cmax = 85Ø;        (*size of code*)
    lmax =   7;        (*maximum level*)
    smax = 6ØØ;        (*size of string-table*)
    ermax = 58;        (*max error no.*)
    omax =  63;        (*highest order code*)
    xmax = 131Ø71;  (*2**17 - 1*)
    nmax = 28147497671Ø655;   (*2**48-1*)
    lineleng = 136; (*output line length*)
    linelimit = 2ØØ;
    stacksize = 15ØØ;

type symbol = (intcon,realcon,charcon,string,
               notsy,plus,minus,times,idiv,rdiv,imod,andsy,orsy,
               eql,neq,gtr,geq,lss,leq,
               lparent,rparent,lbrack,rbrack,comma,semicolon,period,
               colon,becomes,constsy,typesy,varsy,functionsy,
               proceduresy,arraysy,recordsy,programsy,ident,
               beginsy,ifsy,casesy,repeatsy,whilesy,forsy,
               endsy,elsesy,untilsy,ofsy,dosy,tosy,downtosy,thensy);

    index  = -xmax .. +xmax;
    alfa = packed array [1..alng] of char;
    object = (konstant,variable,type1,prozedure,funktion);
    types  = (notyp,ints,reals,bools,chars,arrays,records);
    symset = set of symbol;
```

```
     typset = set of types;
     item   = record
                 typ: types; ref: index;
              end ;
     order  = packed record
                 f: -omax..+omax;
                 x: -lmax..+lmax;
                 y: -nmax..+nmax;
              end ;

var  sy: symbol;                (*last symbol read by insymbol*)
     id: alfa;                  (*identifier from insymbol*)
     inum: integer;             (*integer from insymbol*)
     rnum: real;                (*real number from insymbol*)
     sleng: integer;            (*string length*)
     ch: char;                  (*last character read from source program*)
     line: array [1..llng] of char;
     cc: integer;               (*character counter*)
     lc: integer;               (*program location counter*)
     ll: integer;               (*length of current line*)
     errs: set of Ø..ermax;
     errpos: integer;
     progname: alfa;
     iflag, oflag: boolean;
     constbegsys,typebegsys,blockbegsys,facbegsys,statbegsys: symset;
     key: array [1..nkw] of alfa;
     ksy: array [1..nkw] of symbol;
     sps: array [char] of symbol;  (*special symbols*)

     t,a,b,sx,c1,c2: integer;  (*indices to tables*)
     stantyps: typset;
     display: array [Ø .. lmax] of integer;

     tab:     array [Ø .. tmax] of      (*identifier table*)
                packed record
                  name: alfa;   link: index;
                  obj: object;  typ: types;
                  ref: index;   normal: boolean;
                  lev: Ø .. lmax; adr: integer;
                end ;
     atab:    array [1 .. amax] of      (*array-table*)
                packed record
                  inxtyp, eltyp: types;
                  elref, low, high, elsize, size: index;
                end ;
     btab:    array [1 .. bmax] of    (*block-table*)
                packed record
                   last, lastpar, psize, vsize: index
                end ;
     stab:    packed array [Ø..smax] of char;  (*string table*)
     rconst:  array [1 .. c2max] of real;
     code:    array [Ø .. cmax] of order;
```

```
procedure errormsg;
   var k: integer;
       msg: array [Ø..ermax] of alfa;
begin
  msg[ Ø] := 'undef id    '; msg[ 1] := 'multi def   ';
  msg[ 2] := 'identifier  '; msg[ 3] := 'program     ';
  msg[ 4] := ')           '; msg[ 5] := ':           ';
  msg[ 6] := 'syntax      '; msg[ 7] := 'ident, var  ';
  msg[ 8] := 'of          '; msg[ 9] := '(           ';
  msg[1Ø] := 'id, array   '; msg[11] := '[           ';
  msg[12] := ']           '; msg[13] := '..          ';
  msg[14] := ';           '; msg[15] := 'func. type  ';
  msg[16] := '=           '; msg[17] := 'boolean     ';
  msg[18] := 'convar typ  '; msg[19] := 'type        ';
  msg[2Ø] := 'prog.param  '; msg[21] := 'too big     ';
  msg[22] := '.           '; msg[23] := 'typ (case)  ';
  msg[24] := 'character   '; msg[25] := 'const id    ';
  msg[26] := 'index type  '; msg[27] := 'indexbound  ';
  msg[28] := 'no array    '; msg[29] := 'type id     ';
  msg[3Ø] := 'undef type  '; msg[31] := 'no record   ';
  msg[32] := 'boole type  '; msg[33] := 'arith type  ';
  msg[34] := 'integer     '; msg[35] := 'types       ';
  msg[36] := 'param type  '; msg[37] := 'variab id   ';
  msg[38] := 'string      '; msg[39] := 'no.of pars  ';
  msg[4Ø] := 'type        '; msg[41] := 'type        ';
  msg[42] := 'real type   '; msg[43] := 'integer     ';
  msg[44] := 'var, const  '; msg[45] := 'var, proc   ';
  msg[46] := 'types (:=)  '; msg[47] := 'typ (case)  ';
  msg[48] := 'type        '; msg[49] := 'store ovfl  ';
  msg[5Ø] := 'constant    '; msg[51] := ':=          ';
  msg[52] := 'then        '; msg[53] := 'until       ';
  msg[54] := 'do          '; msg[55] := 'to downto   ';
  msg[56] := 'begin       '; msg[57] := 'end         ';
  msg[58] := 'factor      ';
  k := Ø; writeln; writeln(' key words');
  while errs <> [] do
  begin while not (k in errs) do k := k+1;
        writeln(k,'   ',msg[k]); errs := errs - [k]
  end
end (*errormsg*) ;

procedure nextch;   (*read next character; process line end*)
begin if cc = ll then
      begin if eos(input) then
            begin writeln;
               writeln(' program incomplete');
               errormsg; goto 99
            end ;
         if errpos <> Ø then
            begin writeln; errpos := Ø
            end ;
         write( lc:5, '  ');
         ll := Ø; cc := Ø;
         while not 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 (*nextch*) ;

procedure error(n: integer);
begin if errpos = 0 then write(' ****');
   if cc > errpos then
      begin write(' ': cc-errpos, '↑', n:2);
         errpos := cc+3; errs := errs + [n]
      end
end (*error*) ;

procedure fatal(n: integer);
   var msg: array [1..7] of alfa;
begin writeln; errormsg;
   msg[ 1] := 'identifier'; msg[ 2] := 'procedures';
   msg[ 3] := 'reals     '; msg[ 4] := 'arrays    ';
   msg[ 5] := 'levels    '; msg[ 6] := 'code      ';
   msg[ 7] := 'strings   ';
   writeln(' compiler table for ', msg[n], ' is too small');
   goto 99    (* terminate compilation*)
end (*fatal*) ;

procedure insymbol;           (*reads next symbol*)
   label 1,2,3;
   var i,j,k,e: integer;

   procedure readscale;
      var s, sign: integer;
   begin nextch; sign := 1; s := 0;
      if ch = '+' then nextch else
      if ch = '-' then begin nextch; sign := -1 end ;
      while ch in ['0'..'9'] do
         begin s := 10*s + ord(ch) - ord('0'); nextch
         end ;
      e := s*sign + e
   end (*readscale*) ;

   procedure adjustscale;
      var s: integer; d,t: real;
   begin if k+e > emax then error(21) else
         if k+e < emin then rnum := 0 else
     begin s := abs(e); t := 1.0; d := 10.0;
       repeat
         while not odd(s) do
            begin s := s div 2; d := sqr(d)
            end ;
         s := s-1; t := d*t
       until s = 0;
```

```
         if e >= 0 then rnum := rnum*t else rnum := rnum/t
       end
     end (*adjustscale*) ;

  begin (*insymbol*)
  1: while ch = ' ' do nextch;
     if ch in ['a'..'z'] then
     begin (*word*)  k := 0; id := '          ';
        repeat if k < alng then
                 begin k := k+1; id[k] := ch
                 end ;
           nextch
        until not (ch in ['a'..'z','0'..'9']);
        i := 1; j := nkw;   (*binary search*)
        repeat k := (i+j) div 2;
           if id <= key[k] then j := k-1;
           if id >= key[k] then i := k+1
        until i > j;
        if i-1 > j then sy := ksy[k] else sy := ident
     end else
     if ch in ['0'..'9'] then
     begin (*number*) k := 0; inum := 0; sy := intcon;
        repeat inum := inum*10 + ord(ch) - ord('0');
           k := k+1; nextch
        until not (ch in ['0'..'9']);
        if (k > kmax) or (inum > nmax) then
          begin error(21); inum := 0; k := 0
          end ;
        if ch = '.' then
        begin nextch;
           if ch = '.' then ch := ':' else
              begin sy := realcon; rnum := inum; e := 0;
                 while ch in ['0'..'9'] do
                 begin e := e-1;
                    rnum := 10.0*rnum + (ord(ch)-ord('0')); nextch
                 end ;
                 if ch = 'e' then readscale;
                 if e <> 0 then adjustscale
              end
        end else
        if ch = 'e' then
        begin sy := realcon; rnum := inum; e := 0;
           readscale; if e <> 0 then adjustscale
        end ;
     end else
     case ch of
  ':' : begin nextch;
            if ch = '=' then
              begin sy := becomes; nextch
              end  else sy := colon
        end ;
  '<' : begin nextch;
           if ch = '=' then begin sy := leq; nextch end else
         if ch = '>' then begin sy := neq; nextch end else sy := lss
      end ;
'>' : begin nextch;
         if ch = '=' then begin sy := geq; nextch end else sy := gtr
      end ;
'.' : begin nextch;
         if ch = '.' then
            begin sy := colon; nextch
            end  else sy := period
      end ;
'''': begin k := Ø;
    2:  nextch;
        if ch = '''' then
          begin nextch; if ch <> '''' then goto 3
          end ;
        if sx+k = smax then fatal(7);
        stab[sx+k] := ch; k := k+1;
        if cc = 1 then
          begin (*end of line*) k := Ø;
          end
        else goto 2;
    3:  if k = 1 then
           begin sy := charcon; inum := ord(stab[sx])
           end else
        if k = Ø then
           begin error(38); sy := charcon; inum := Ø
           end else
           begin sy := string; inum := sx; sleng := k; sx := sx+k
           end
      end ;
'(' : begin nextch;
         if ch <> '*' then sy := lparent else
         begin (*comment*) nextch;
            repeat
               while ch <> '*' do nextch;
               nextch
            until ch = ')';
            nextch; goto 1
         end
      end ;
'+', '-', '*', '/', ')', '=', ',', '[', ']', '#', '&', ';' :
      begin sy := sps[ch]; nextch
      end ;
'$', '%', '@', '\', '~', '{', '}', '↑' :
      begin error(24); nextch; goto 1
      end
   end
end (*insymbol*) ;
```

```
procedure enter(x0: alfa; x1: object;
                x2: types; x3: integer);
begin t := t+1;   (*enter standard identifier*)
   with tab[t] do
   begin name := x0; link := t-1; obj := x1;
      typ := x2; ref := 0; normal := true;
      lev := 0; adr := x3
   end
end (*enter*) ;

procedure enterarray(tp: types; l,h: integer);
begin if l > h then error(27);
   if (abs(l)>xmax) or (abs(h)>xmax) then
      begin error(27); l := 0; h := 0;
      end ;
   if a = amax then fatal(4) else
      begin a := a+1;
        with atab[a] do
           begin inxtyp := tp; low := l; high := h
           end
      end
end (*enterarray*) ;

procedure enterblock;
begin if b = bmax then fatal(2) else
      begin b := b+1; btab[b].last := 0; btab[b].lastpar := 0
      end
end (*enterblock*) ;

procedure enterreal(x: real);
begin if c2 = c2max-1 then fatal(3) else
      begin rconst[c2+1] := x; c1 := 1;
         while rconst[c1] <> x do  c1 := c1+1;
         if c1 > c2 then c2 := c1
      end
end (*enterreal*) ;

procedure emit(fct: integer);
begin if lc = cmax then fatal(6);
   code[lc].f := fct; lc := lc+1
end (*emit*) ;

procedure emit1(fct,b: integer);
begin if lc = cmax then fatal(6);
   with code[lc] do
      begin f := fct; y := b end ;
   lc := lc+1
end (*emit1*) ;

procedure emit2(fct,a,b: integer);
begin if lc = cmax then fatal(6);
   with code[lc] do
```

```
     begin f := fct; x := a; y := b end ;
   lc := lc+1
end (*emit2*) ;

procedure printtables;
   var i: integer; o: order;
begin
   writeln('0identifiers     link  obj  typ  ref  nrm  lev  adr');
   for i := btab[1].last +1 to t do
      with tab[i] do
      writeln(i,' ',name,link:5, ord(obj):5, ord(typ):5, ref:5,
            ord(normal):5, lev:5, adr:5);
   writeln('0blocks    last lpar psze vsze');
   for i := 1 to b do
      with btab[i] do
      writeln(i, last:5, lastpar:5, psize:5, vsize:5);
   writeln('0arrays    xtyp etyp eref  low high elsz size');
   for i := 1 to a do
      with atab[i] do
      writeln(i, ord(inxtyp):5, ord(eltyp):5,
              elref:5, low:5, high:5, elsize:5, size:5);
   writeln('0code:');
   for i := 0 to lc-1 do
   begin if i mod 5 = 0 then
         begin writeln; write(i:5)
         end ;
      o := code[i]; write(o.f:5);
      if o.f < 31 then
        if o.f < 4 then write(o.x:2, o.y:5)
                   else write(o.y:7)
      else write('        ');
      write(',')
   end ;
   writeln
end (*printtables*) ;


procedure block(fsys: symset; isfun: boolean; level: integer);
   type conrec =
      record case tp: types of
         ints,chars,bools: (i: integer);
         reals: (r: real)
      end ;

   var dx: integer;     (*data allocation index*)
       prt: integer;    (*t-index of this procedure*)
       prb: integer;    (*b-index of this procedure*)
       x: integer;

   procedure skip(fsys: symset; n: integer);
   begin error(n);
      while not (sy in fsys) do insymbol
   end (*skip*) ;
```

```
procedure test(s1,s2: symset; n: integer);
begin if not (sy in s1) then
      skip(s1+s2,n)
end (*test*) ;

procedure testsemicolon;
begin
  if sy = semicolon then insymbol else
  begin error(14);
    if sy in [comma,colon] then insymbol
  end ;
  test([ident]+blockbegsys, fsys, 6)
end (*testsemicolon*) ;

procedure enter(id: alfa; k: object);
   var j,l: integer;
begin if t = tmax then fatal(1) else
      begin tab[Ø].name := id;
         j := btab[display[level]].last;  l := j;
         while tab[j].name <> id do  j := tab[j].link;
         if j <> Ø then error(1) else
         begin t := t+1;
           with tab[t] do
           begin name := id; link := l;
            obj := k; typ := notyp; ref := Ø; lev := level;
            adr := Ø
           end ;
           btab[display[level]].last := t
         end
      end
end (*enter*) ;

function loc(id: alfa): integer;
   var i,j: integer;     (*locate id in table*)
begin i := level; tab[Ø].name := id;   (*sentinel*)
   repeat j := btab[display[i]].last;
      while tab[j].name <> id do  j := tab[j].link;
      i := i-1;
   until (i<Ø) or (j<>Ø);
   if j = Ø then error(Ø);  loc := j
end (*loc*) ;

procedure entervariable;
begin if sy = ident then
        begin enter(id,variable); insymbol
        end
      else error(2)
end (*entervariable*) ;

procedure constant(fsys: symset; var c: conrec);
  var x, sign: integer;
begin c.tp := notyp; c.i := Ø;
```

```
    test(constbegsys, fsys, 5Ø);
    if sy in constbegsys then
    begin
        if sy = charcon then
          begin c.tp := chars; c.i := inum; insymbol
          end
        else
          begin sign := 1;
            if sy in [plus,minus] then
              begin if sy = minus then sign := -1;
                insymbol
              end ;
            if sy = ident then
              begin x := loc(id);
                if x <> Ø then
                  if tab[x].obj <> konstant then error(25) else
                  begin c.tp := tab[x].typ;
                    if c.tp = reals
                         then c.r := sign*rconst[tab[x].adr]
                         else c.i := sign*tab[x].adr
                  end ;
                insymbol
              end
            else
            if sy = intcon then
              begin c.tp := ints; c.i := sign*inum; insymbol
              end else
            if sy = realcon then
              begin c.tp := reals; c.r := sign*rnum; insymbol
              end else skip(fsys,5Ø)
          end;
        test(fsys, [], 6)
      end
  end (*constant*) ;

  procedure typ(fsys: symset; var tp: types; var rf, sz: integer);
    var x: integer;
        eltp: types; elrf: integer;
        elsz, offset, tØ,t1: integer;

    procedure arraytyp(var aref,arsz: integer);
       var eltp: types;
           low, high: conrec;
           elrf, elsz: integer;
    begin constant([colon,rbrack,rparent,ofsy]+fsys, low);
       if low.tp = reals then
          begin error(27); low.tp := ints; low.i := Ø
          end ;
       if sy = colon then insymbol else error(13);
       constant([rbrack,comma,rparent,ofsy]+fsys, high);
       if high.tp <> low.tp then
          begin error(27); high.i := low.i
          end ;
        enterarray(low.tp, low.i, high.i); aref := a;
        if sy = comma then
           begin insymbol; eltp := arrays; arraytyp(elrf,elsz)
           end else
        begin
           if sy = rbrack then insymbol else
              begin error(12);
                 if sy = rparent then insymbol
              end ;
           if sy = ofsy then insymbol else error(8);
           typ(fsys,eltp,elrf,elsz)
        end ;
        with atab[aref] do
        begin arsz := (high-low+1)*elsz; size := arsz;
           eltyp := eltp; elref := elrf; elsize := elsz
        end ;
     end (*arraytyp*) ;

   begin (*typ*) tp := notyp; rf := 0; sz := 0;
     test(typebegsys, fsys, 10);
     if sy in typebegsys then
       begin
         if sy = ident then
         begin x := loc(id);
           if x <> 0 then
           with tab[x] do
             if obj <> type1 then error(29) else
             begin tp := typ; rf := ref; sz := adr;
               if tp = notyp then error(30)
             end ;
           insymbol
         end else
         if sy = arraysy then
         begin insymbol;
             if sy = lbrack then insymbol else
                begin error(11);
                   if sy = lparent then insymbol
                end ;
             tp := arrays; arraytyp(rf,sz)
         end else
         begin (*records*) insymbol;
           enterblock; tp := records; rf := b;
           if level = lmax then fatal(5);
           level := level+1; display[level] := b; offset := 0;
           while sy <> endsy do
           begin (*field section*)
             if sy = ident then
             begin t0 := t; entervariable;
               while sy = comma do
                 begin insymbol; entervariable
                 end ;
               if sy = colon then insymbol else error(5);
               t1 := t;
```

```
            typ(fsys+[semicolon,endsy,comma,ident],
                eltp,elrf,elsz);
            while t0 < t1 do
            begin t0 := t0+1;
              with tab[t0] do
              begin typ := eltp; ref := elrf; normal := true;
                adr := offset; offset := offset + elsz
              end
            end
          end ;
          if sy <> endsy then
          begin if sy = semicolon then insymbol else
                begin error(14);
                  if sy = comma then insymbol
                end ;
             test([ident,endsy,semicolon], fsys, 6)
          end
        end ;
        btab[rf].vsize := offset; sz := offset;
        btab[rf].psize := 0; insymbol; level := level-1
      end ;
      test(fsys, [], 6)
    end
end (*typ*) ;

procedure parameterlist;     (*formal parameter list*)
   var tp: types;
       rf, sz, x, t0: integer;
       valpar: boolean;
begin insymbol; tp := notyp; rf := 0; sz := 0;
  test([ident, varsy], fsys+[rparent], 7);
  while sy in [ident,varsy] do
    begin if sy <> varsy then valpar := true else
            begin insymbol; valpar := false
            end ;
      t0 := t; entervariable;
      while sy = comma do
         begin insymbol; entervariable;
         end ;
      if sy = colon then
        begin insymbol;
          if sy <> ident then error(2) else
          begin x := loc(id); insymbol;
            if x <> 0 then
            with tab[x] do
              if obj <> type1 then error(29) else
                begin tp := typ; rf := ref;
                  if valpar then sz := adr else sz := 1
                end :
          end ;
          test([semicolon,rparent], [comma,ident]+fsys, 14)
        end
      else error(5);
      while tØ < t do
      begin tØ := tØ+1;
        with tab[tØ] do
        begin typ := tp; ref := rf;
            normal := valpar; adr := dx; lev := level;
            dx := dx + sz
        end
      end ;
      if sy <> rparent then
      begin if sy = semicolon then insymbol else
            begin error(14);
              if sy = comma then insymbol
            end ;
         test([ident,varsy], [rparent]+fsys, 6)
      end
    end (*while*) ;
  if sy = rparent then
    begin insymbol;
      test([semicolon,colon], fsys, 6)
    end
  else error(4)
end (*parameterlist*) ;

procedure constantdeclaration;
  var c: conrec;
begin insymbol;
  test([ident], blockbegsys, 2);
  while sy = ident do
    begin enter(id,konstant); insymbol;
      if sy = eql then insymbol else
         begin error(16);
            if sy = becomes then insymbol
         end ;
      constant([semicolon,comma,ident]+fsys,c);
      tab[t].typ := c.tp; tab[t].ref := Ø;
      if c.tp = reals then
        begin enterreal(c.r); tab[t].adr := c1 end
      else tab[t].adr := c.i;
      testsemicolon
    end
end (*constantdeclaration*) ;

procedure typedeclaration;
  var tp: types; rf, sz, t1: integer;
begin insymbol;
  test([ident], blockbegsys, 2);
  while sy = ident do
    begin enter(id,type1); t1 := t; insymbol;
      if sy = eql then insymbol else
         begin error(16);
            if sy = becomes then insymbol
         end ;
      typ([semicolon,comma,ident]+fsys, tp, rf, sz);
```

```
        with tab[t1] do
          begin typ := tp; ref := rf; adr := sz
          end ;
        testsemicolon
      end
    end (*typedeclaration*) ;

    procedure variabledeclaration;
      var t0, t1, rf, sz: integer;
          tp: types;
    begin insymbol;
      while sy = ident do
      begin t0 := t; entervariable;
        while sy = comma do
          begin insymbol; entervariable;
          end ;
        if sy = colon then insymbol else error(5);
        t1 := t;
        typ([semicolon,comma,ident]+fsys, tp, rf, sz);
        while t0 < t1 do
        begin t0 := t0+1;
          with tab[t0] do
          begin typ := tp; ref := rf;
            lev := level; adr := dx; normal := true;
            dx := dx + sz
          end
        end ;
        testsemicolon
      end
    end (*variabledeclaration*) ;

    procedure procdeclaration;
       var isfun: boolean;
    begin isfun := sy = functionsy; insymbol;
      if sy <> ident then
         begin  error(2); id := '          '
         end ;
      if isfun then enter(id,funktion) else enter(id,prozedure);
      tab[t].normal := true;
      insymbol; block([semicolon]+fsys, isfun, level+1);
      if sy = semicolon then insymbol else error(14);
      emit(32+ord(isfun))    (*exit*)
    end (*proceduredeclaration*) ;



    procedure statement(fsys: symset);
       var i: integer; x: item;
       procedure expression(fsys: symset; var x: item); forward;

       procedure selector(fsys: symset; var v:item);
          var x: item; a,j: integer;
       begin (*sy in [lparent, lbrack, period]*)
```

```
  repeat
    if sy = period then
    begin insymbol;  (*field selector*)
      if sy <> ident then error(2) else
      begin
        if v.typ <> records then error(31) else
        begin (*search field identifier*)
          j := btab[v.ref] .last; tab[0].name := id;
          while tab[j].name <> id do j := tab[j].link;
          if j = 0 then error(0);
          v.typ := tab[j].typ; v.ref := tab[j].ref;
          a := tab[j].adr; if a <> 0 then emit1(9,a)
        end ;
        insymbol
      end
    end else
    begin (*array selector*)
      if sy <> lbrack then error(11);
      repeat insymbol;
        expression(fsys+[comma,rbrack], x);
        if v.typ <> arrays then error(28) else
          begin a := v.ref;
            if atab[a].inxtyp <> x.typ then error(26) else
          if atab[a].elsize = 1 then emit1(20,a)
                                else emit1(21,a);
            v.typ := atab[a].eltyp; v.ref := atab[a].elref
          end
      until sy <> comma;
      if sy = rbrack then insymbol else
        begin error(12); if sy = rparent then insymbol
        end
    end
  until not (sy in [lbrack,lparent,period]);
  test(fsys, [], 6)
end (*selector*) ;

procedure call(fsys: symset; i: integer);
   var x: item;
       lastp, cp, k: integer;
begin emit1(18,i);  (*mark stack*)
  lastp := btab[tab[i].ref].lastpar; cp := i;
  if sy = lparent then
  begin (*actual parameter list*)
    repeat insymbol;
      if cp >= lastp then error(39) else
      begin cp := cp+1;
        if tab[cp].normal then
        begin (*value parameter*)
          expression(fsys+[comma,colon,rparent], x);
          if x.typ=tab[cp].typ then
            begin
              if x.ref <> tab[cp].ref then error(36) else
        if x.typ = arrays then emit1(22,atab[x.ref].size) else
```

```
                 if x.typ = records then emit1(22,btab[x.ref].vsize)

                  end else
                if (x.typ=ints) and (tab[cp].typ=reals) then
                   emit1(26,Ø) else
                   if x.typ<>notyp then error(36);
             end else
             begin (*variable parameter*)
               if sy <> ident then error(2) else
               begin k := loc(id); insymbol;
                 if k <> Ø then
                 begin if tab[k].obj <> variable then error(37);
                   x.typ := tab[k].typ; x.ref := tab[k].ref;
                   if tab[k].normal
                      then emit2(Ø,tab[k].lev,tab[k].adr)
                      else emit2(1,tab[k].lev,tab[k].adr);
                   if sy in [lbrack,lparent,period] then
                      selector(fsys+[comma,colon,rparent], x);
                   if (x.typ<>tab[cp].typ) or (x.ref<>tab[cp].ref)
                   then error(36)
                 end
               end
             end
           end ;
           test([comma,rparent], fsys, 6)
         until sy <> comma;
         if sy = rparent then insymbol else error(4)
       end ;
       if cp < lastp then error(39); (*too few actual parameters*)
       emit1(19, btab[tab[i].ref].psize-1);
       if tab[i].lev < level then emit2(3, tab[i].lev, level)
     end (*call*) ;

     function resulttype(a,b: types): types;
     begin
       if (a>reals) or (b>reals) then
         begin error(33); resulttype := notyp
         end else
       if (a=notyp) or (b=notyp) then resulttype := notyp else
       if a=ints then
         if b=ints then resulttype := ints else
           begin resulttype := reals; emit1(26,1)
           end
       else
         begin resulttype := reals;
           if b=ints then emit1(26,Ø)
         end
     end (*resulttype*) ;

     procedure expression;
       var y:item; op:symbol;

       procedure simpleexpression(fsys:symset; var x:item);
```

```
          var y:item; op:symbol;

          procedure term(fsys:symset; var x:item);
            var y:item; op:symbol; ts:typset;

            procedure factor(fsys:symset; var x:item);
              var i,f: integer;

              procedure standfct(n: integer);
                 var ts: typset;
              begin (*standard function no. n*)
                if sy = lparent then insymbol else error(9);
                if n < 17 then
                  begin expression(fsys+[rparent],x);
                    case n of
(*abs,sqr*)         Ø,2:  begin ts := [ints,reals];
                            tab[i].typ := x.typ;
                            if x.typ = reals then n := n+1
                          end ;
(*odd,chr*)         4,5:  ts := [ints];
(*ord*)             6:    ts := [ints,bools,chars];
(*succ,pred*)       7,8:  ts := [chars];
(*round,trunc*)     9,1Ø,11,12,13,14,15,16:
(*sin,cos,...*)           begin ts := [ints,reals];
                            if x.typ = ints then emit1(26,Ø)
                          end ;
                    end ;
                    if x.typ in ts then emit1(8,n) else
                    if x.typ <> notyp then error(48);
                  end else
(*eof,eoln*)      begin (*n in [17,18]*)
                    if sy <> ident then error(2) else
                  if id <> 'input      ' then error(Ø) else insymbol;

                    emit1(8,n);
                  end ;
                x.typ := tab[i].typ;
                if sy = rparent then insymbol else error(4)
              end (*standfct*) ;

            begin (*factor*) x.typ := notyp; x.ref := Ø;
              test(facbegsys, fsys, 58);
              while sy in facbegsys do
                begin
                  if sy = ident then
                  begin i := loc(id); insymbol;
                    with tab[i] do
                    case obj of
              konstant: begin x.typ := typ; x.ref := Ø;
                          if x.typ = reals then
                            emit1(25,adr) else
                            emit1(24,adr)
                        end ;
            variable: begin x.typ := typ; x.ref := ref;
                        if sy in [lbrack,lparent,period] then
                          begin if normal then f := 0 else f := 1;
                            emit2(f, lev, adr);
                            selector(fsys,x);
                            if x.typ in stantyps then emit(34)
                          end else
                          begin
                            if x.typ in stantyps then
                              if normal then f := 1 else f := 2
                            else
                              if normal then f := 0 else f := 1;
                            emit2(f, lev, adr)
                          end
                      end ;
            type1, prozedure:    error(44);
            funktion :begin x.typ := typ;
                        if lev <> 0 then call(fsys, i)
                             else standfct(adr)
                      end
                end (*case,with*)
              end else
              if sy in [charcon,intcon,realcon] then
               begin
                 if sy = realcon then
                 begin x.typ := reals; enterreal(rnum);
                   emit1(25, c1)
                 end else
                 begin if sy = charcon then x.typ := chars
                                       else x.typ := ints;
                   emit1(24, inum)
                 end ;
                 x.ref := 0; insymbol
               end else
              if sy = lparent then
               begin insymbol; expression(fsys+[rparent], x);
                 if sy = rparent then insymbol else error(4)
               end else
              if sy = notsy then
               begin insymbol; factor(fsys,x);
                 if x.typ=bools then emit(35) else
                   if x.typ<>notyp then error(32)
               end ;
              test(fsys, facbegsys, 6)
             end (*while*)
         end (*factor*) ;
       begin (*term*)
         factor(fsys+[times,rdiv,idiv,imod,andsy], x);
         while sy in [times,rdiv,idiv,imod,andsy] do
           begin op := sy; insymbol;
             factor(fsys+[times,rdiv,idiv,imod,andsy], y);
             if op = times then
             begin x.typ := resulttype(x.typ, y.typ);
```

```
          case x.typ of
            notyp: ;
            ints : emit(57);
            reals: emit(60);
          end
        end else
        if op = rdiv then
        begin
          if x.typ = ints then
            begin emit1(26,1); x.typ := reals
            end ;
          if y.typ = ints then
            begin emit1(26,0); y.typ := reals
            end ;
          if (x.typ=reals) and (y.typ=reals) then
            emit(61) else
            begin if (x.typ<>notyp) and (y.typ<>notyp) then
                    error(33);
                  x.typ := notyp
            end
        end else
        if op = andsy then
        begin if (x.typ=bools) and (y.typ=bools) then
                 emit(56) else
              begin if (x.typ<>notyp) and (y.typ<>notyp)
                  then error(32);
                 x.typ := notyp
              end
        end else
        begin (*op in [idiv,imod]*)
          if (x.typ=ints) and (y.typ=ints) then
            if op=idiv then emit(58)
                       else emit(59) else
            begin if (x.typ<>notyp) and (y.typ<>notyp) then
                     error(34);
                  x.typ := notyp
            end
        end
      end
  end (*term*) ;
begin (*simpleexpression*)
  if sy in [plus,minus] then
    begin op := sy; insymbol;
      term( fsys+[plus,minus], x);
      if x.typ > reals then error(33) else
        if op = minus then emit(36)
    end else
  term( fsys+[plus,minus,orsy], x);
  while sy in [plus,minus,orsy] do
    begin op := sy; insymbol;
       term( fsys+[plus,minus,orsy], y);
       if op = orsy then
       begin
```

```
              if (x.typ=bools) and (y.typ=bools) then emit(51) else
                  begin if (x.typ<>notyp) and (y.typ<>notyp) then
                           error(32);
                        x.typ := notyp
                  end
              end else
              begin x.typ := resulttype(x.typ, y.typ);
                case x.typ of
                  notyp: ;
                  ints : if op = plus then emit(52)
                                      else emit(53);
                  reals: if op = plus then emit(54)
                                      else emit(55)
                end
              end
           end
       end (*simpleexpression*) ;
     begin (*expression*)
       simpleexpression( fsys+[eql,neq,lss,leq,gtr,geq], x);
       if sy in [eql,neq,lss,leq,gtr,geq] then
         begin op := sy; insymbol;
             simpleexpression( fsys, y);
             if (x.typ in [ notyp,ints,bools,chars]) and
                (x.typ = y.typ) then
               case op of
                 eql: emit(45);
                 neq: emit(46);
                 lss: emit(47);
                 leq: emit(48);
                 gtr: emit(49);
                 geq: emit(50);
               end else
             begin if x.typ = ints then
                      begin x.typ := reals; emit1(26,1)
                      end else
                    if y.typ = ints then
                      begin y.typ := reals; emit1(26,0)
                      end ;
               if (x.typ=reals) and (y.typ=reals) then
                 case op of
                   eql: emit(39);
                   neq: emit(40);
                   lss: emit(41);
                   leq: emit(42);
                   gtr: emit(43);
                   geq: emit(44);
                 end
               else error(35)
             end ;
             x.typ := bools
         end
     end (*expression*) ;
```

```
procedure assignment(lv,ad: integer);
   var x,y: item; f: integer;
   (*tab[i].obj in [variable,prozedure]*)
begin x.typ := tab[i].typ; x.ref := tab[i].ref;
  if tab[i].normal then f := Ø else f := 1;
  emit2(f, lv, ad);
  if sy in [lbrack,lparent,period] then
     selector([becomes,eql]+fsys, x);
  if sy = becomes then insymbol else
    begin error(51); if sy = eql then insymbol
    end ;
  expression(fsys, y);
  if x.typ = y.typ then
    if x.typ in stantyps then emit(38) else
    if x.ref <> y.ref then error(46) else
    if x.typ = arrays then emit1(23, atab[x.ref].size)
                      else emit1(23, btab[x.ref].vsize)
  else
  if (x.typ=reals) and (y.typ=ints) then
    begin emit1(26,Ø); emit(38)
    end else
    if (x.typ<>notyp) and (y.typ<>notyp) then error(46)
end (*assignment*) ;

procedure compoundstatement;
begin insymbol;
  statement([semicolon,endsy]+fsys);
  while sy in [semicolon]+statbegsys do
  begin if sy = semicolon then insymbol else error(14);
    statement([semicolon,endsy]+fsys)
  end ;
  if sy = endsy then insymbol else error(57)
end (*compoundstatemenet*) ;

procedure ifstatement;
   var x: item; lc1,lc2: integer;
begin insymbol;
  expression(fsys+[thensy,dosy], x);
  if not (x.typ in [bools,notyp]) then error(17);
  lc1 := lc; emit(11);  (*jmpc*)
  if sy = thensy then insymbol else
    begin error(52); if sy = dosy then insymbol
    end ;
  statement(fsys+[elsesy]);
  if sy = elsesy then
    begin insymbol; lc2 := lc; emit(1Ø);
      code[lc1].y := lc; statement(fsys); code[lc2].y := lc
    end
  else code[lc1].y := lc
end (*ifstatement*) ;

procedure casestatement;
  var x: item;
```

```
        i,j,k,lc1: integer;
        casetab: array [1..csmax] of
                   packed record val, lc: index end ;
        exittab: array [1..csmax] of integer;

   procedure caselabel;
     var lab: conrec; k: integer;
   begin constant(fsys+[comma,colon], lab);
     if lab.tp <> x.typ then error(47) else
     if i = csmax then fatal(6) else
       begin i := i+1; k := 0;
         casetab[i].val := lab.i; casetab[i].lc := lc;
         repeat k := k+1 until casetab[k].val = lab.i;
         if k < i then error(1);   (*multiple definition*)
       end
   end (*caselabel*) ;

   procedure onecase;
   begin if sy in constbegsys then
     begin caselabel;
       while sy = comma do
         begin insymbol; caselabel
         end ;
       if sy = colon then insymbol else error(5);
       statement([semicolon,endsy]+fsys);
       j := j+1; exittab[j] := lc; emit(10)
     end
   end (*onecase*) ;

 begin insymbol; i := 0; j := 0;
   expression(fsys+[ofsy,comma,colon], x);
   if not (x.typ in [ints,bools,chars,notyp]) then error(23);
   lc1 := lc; emit(12);  (*jmpx*)
   if sy = ofsy then insymbol else error(8);
   onecase;
   while sy = semicolon do
     begin insymbol; onecase
     end ;
   code[lc1].y := lc;
   for k := 1 to i do
     begin emit1(13,casetab[k].val); emit1(13,casetab[k].lc)
     end ;
   emit1(10,0);
   for k := 1 to j do code[exittab[k]].y := lc;
   if sy = endsy then insymbol else error(57)
 end (*casestatement*) ;

 procedure repeatstatement;
    var x: item; lc1: integer;
 begin lc1 := lc;
   insymbol; statement([semicolon,untilsy]+fsys);
   while sy in [semicolon]+statbegsys do
   begin if sy = semicolon then insymbol else error(14);
```

```
      statement([semicolon,untilsy]+fsys)
    end ;
    if sy = untilsy then
      begin insymbol; expression(fsys, x);
        if not (x.typ in [bools,notyp]) then error(17);
        emit1(11,lc1)
      end
    else error(53)
  end (*repeatstatement*) ;

  procedure whilestatement;
     var x: item; lc1,lc2: integer;
  begin insymbol; lc1 := lc;
    expression(fsys+[dosy], x);
    if not (x.typ in [bools,notyp]) then error(17);
    lc2 := lc; emit(11);
    if sy = dosy then insymbol else error(54);
    statement(fsys); emit1(10,lc1); code[lc2].y := lc
  end (*whilestatement*) ;

  procedure forstatement;
     var cvt: types; x: item;
         i,f,lc1,lc2: integer;
  begin insymbol;
    if sy = ident then
      begin i := loc(id); insymbol;
        if i = 0 then cvt := ints else
        if tab[i].obj = variable then
          begin cvt := tab[i].typ;
            emit2(0, tab[i].lev, tab[i].adr);
            if not (cvt in [notyp,ints,bools,chars])
               then error(18)
          end else
          begin error(37); cvt := ints
          end
      end else skip([becomes,tosy,downtosy,dosy]+fsys, 2);
    if sy = becomes then
      begin insymbol; expression([tosy,downtosy,dosy]+fsys, x);
        if x.typ <> cvt then error(19);
      end else skip([tosy,downtosy,dosy]+fsys, 51);
    f := 14;
    if sy in [tosy, downtosy] then
      begin if sy = downtosy then f := 16;
        insymbol; expression([dosy]+fsys, x);
        if x.typ <> cvt then error(19)
      end else skip([dosy]+fsys, 55);
    lc1 := lc; emit(f);
    if sy = dosy then insymbol else error(54);
    lc2 := lc; statement(fsys);
    emit1(f+1,lc2); code[lc1].y := lc
  end (*forstatement*) ;

  procedure standproc(n: integer);
```

```
      var i,f: integer;
          x,y: item;
   begin
     case n of
1,2: begin (*read*)
       if not iflag then
         begin error(2Ø); iflag := true
         end ;
       if sy = lparent then
       begin
         repeat insymbol;
           if sy <> ident then error(2) else
           begin i := loc(id); insymbol;
             if i <> Ø then
             if tab[i].obj <> variable then error(37) else
             begin x.typ := tab[i].typ; x.ref := tab[i].ref;
               if tab[i].normal then f := Ø else f := 1;
               emit2(f, tab[i].lev, tab[i].adr);
               if sy in [lbrack,lparent,period] then
                 selector(fsys+[comma,rparent], x);
               if x.typ in [ints,reals,chars,notyp] then
                 emit1(27, ord(x.typ)) else error(4Ø)
             end
           end ;
           test([comma,rparent], fsys, 6);
         until sy <> comma;
         if sy = rparent then insymbol else error(4)
       end ;
       if n = 2 then emit(62)
     end ;
3,4: begin (*write*)
       if sy = lparent then
       begin
         repeat insymbol;
           if sy = string then
             begin emit1(24,sleng); emit1(28,inum); insymbol
             end else
           begin expression(fsys+[comma,colon,rparent], x);
             if not (x.typ in stantyps) then error(41);
             if sy = colon then
             begin insymbol;
               expression(fsys+[comma,colon,rparent], y);
               if y.typ <> ints then error(43);
               if sy = colon then
               begin if x.typ <> reals then error(42);
                 insymbol; expression(fsys+[comma,rparent], y)
                 if y.typ <> ints then error(43);
                 emit(37)
               end
               else emit1(3Ø, ord(x.typ))
             end
             else emit1(29, ord(x.typ))
           end
```

```
              until sy <> comma;
              if sy = rparent then insymbol else error(4)
            end ;
            if n = 4 then emit(63)
          end ;
          end (*case*)
        end (*standproc*) ;

      begin (*statement*)
        if sy in statbegsys+[ident] then
            case sy of
              ident:    begin i := loc(id); insymbol;
                          if i <> Ø then
                          case tab[i].obj of
                            konstant, type1: error(45);
                            variable:
                                assignment(tab[i].lev, tab[i].adr);
                            prozedure:
                              if tab[i].lev <> Ø then call(fsys, i)
                                      else standproc(tab[i].adr);
                            funktion:
                              if tab[i].ref = display[level]
                                then assignment(tab[i].lev+1,Ø)
                                else error(45)
                          end
                        end ;
              beginsy:  compoundstatement;
              ifsy:     ifstatement;
              casesy:   casestatement;
              whilesy:  whilestatement;
              repeatsy: repeatstatement;
              forsy:    forstatement;
            end;
          test(fsys, [], 14)
      end (*statement*) ;

  begin (*block*) dx := 5; prt := t;
    if level > lmax then fatal(5);
    test([lparent,colon,semicolon], fsys, 7);
    enterblock; display[level] := b; prb := b;
    tab[prt].typ := notyp; tab[prt].ref := prb;
    if sy = lparent then parameterlist;
    btab[prb].lastpar := t; btab[prb].psize := dx;
    if isfun then
      if sy = colon then
      begin insymbol;   (*function type*)
        if sy = ident then
        begin x := loc(id); insymbol;
          if x <> Ø then
            if tab[x].obj <> type1 then error(29) else
              if tab[x].typ in stantyps
                then tab[prt].typ := tab[x].typ
                else error(15)
      end else skip([semicolon]+fsys, 2)
    end else error(5);
  if sy = semicolon then insymbol else error(14);
  repeat
    if sy = constsy then constantdeclaration;
    if sy = typesy then typedeclaration;
    if sy = varsy then variabledeclaration;
    btab[prb].vsize := dx;
    while sy in [proceduresy,functionsy] do procdeclaration;
    test([beginsy], blockbegsys+statbegsys, 56)
  until sy in statbegsys;
  tab[prt].adr := lc;
  insymbol; statement([semicolon,endsy]+fsys);
  while sy in [semicolon]+statbegsys do
    begin if sy = semicolon then insymbol else error(14);
      statement([semicolon,endsy]+fsys)
    end ;
  if sy = endsy then insymbol else error(57);
  test(fsys+[period], [], 6)
end (*block*) ;


procedure interpret;
  (*global code, tab, btab*)
  var ir: order;        (*instruction buffer*)
      pc: integer;      (*program counter*)
      ps: (run,fin,caschk,divchk,inxchk,stkchk,linchk,
           lngchk,redchk);
      t:  integer;      (*top stack index*)
      b:  integer;      (*base index*)
      lncnt, ocnt, blkcnt, chrcnt: integer;     (*counters*)
      h1,h2,h3,h4: integer;
      fld: array [1..4] of integer;     (*default field widths*)

      display: array [1..lmax] of integer;
      s: array [1..stacksize] of       (*blockmark:             *)
         record case types of          (*   s[b+0] = fct result  *)
           ints:  (i: integer);        (*   s[b+1] = return adr  *)
           reals: (r: real);           (*   s[b+2] = static link *)
           bools: (b: boolean);        (*   s[b+3] = dynamic link*)
           chars: (c: char)            (*   s[b+4] = table index *)
         end ;

begin (*interpret*)
  s[1].i := 0; s[2].i := 0; s[3].i := -1; s[4].i := btab[1].last;
  b := 0; display[1] := 0;
  t := btab[2].vsize - 1; pc := tab[s[4].i].adr;
  ps := run;
  lncnt := 0; ocnt := 0; chrcnt := 0;
  fld[1] := 10; fld[2] := 22; fld[3] := 10; fld[4] := 1;
  repeat ir := code[pc]; pc := pc+1; ocnt := ocnt + 1;
    case ir.f of
  0: begin (*load address*) t := t+1;
```

```
      if t > stacksize then ps := stkchk
        else s[t].i := display[ir.x] + ir.y
    end ;
 1: begin (*load value*) t := t+1;
      if t > stacksize then ps := stkchk
        else s[t] := s[display[ir.x] + ir.y]
    end ;
 2: begin (*load indirect*) t := t+1;
      if t > stacksize then ps := stkchk
        else s[t] := s[s[display[ir.x] + ir.y].i]
    end ;
 3: begin (*update display*)
      h1 := ir.y; h2 := ir.x; h3 := b;
      repeat display[h1] := h3; h1 := h1-1; h3 := s[h3+2].i
      until h1 = h2
    end ;
 8: case ir.y of
     Ø: s[t].i := abs(s[t].i);
     1: s[t].r := abs(s[t].r);
     2: s[t].i := sqr(s[t].i);
     3: s[t].r := sqr(s[t].r);
     4: s[t].b := odd(s[t].i);
     5: begin (* s[t].c := chr(s[t].i); *)
          if (s[t].i < Ø) or (s[t].i > 63) then ps := inxchk
        end ;
     6: (* s[t].i := ord(s[t].c) *);
     7: s[t].c := succ(s[t].c);
     8: s[t].c := pred(s[t].c);
     9: s[t].i := round(s[t].r);
    1Ø: s[t].i := trunc(s[t].r);
    11: s[t].r := sin(s[t].r);
    12: s[t].r := cos(s[t].r);
    13: s[t].r := exp(s[t].r);
    14: s[t].r := ln(s[t].r);
    15: s[t].r := sqrt(s[t].r);
    16: s[t].r := arctan(s[t].r);
    17: begin t := t+1;
          if t > stacksize then ps := stkchk
                           else s[t].b := eos(input)
        end ;
    18: begin t := t+1;
          if t > stacksize then ps := stkchk
                           else s[t].b := eoln(input)
        end ;
    end ;
 9: s[t].i := s[t].i + ir.y;   (*offset*)
1Ø: pc := ir.y;  (*jump*)
11: begin (*conditional jump*)
      if not s[t].b then pc := ir.y;  t := t-1
    end ;
12: begin (*switch*) h1 := s[t].i; t := t-1;
      h2 := ir.y; h3 := Ø;
      repeat if code[h2].f <> 13 then
```

```
                begin h3 := 1; ps := caschk
                end else
              if code[h2].y = h1 then
                begin h3 := 1; pc := code[h2+1].y
                end else
              h2 := h2 + 2
       until h3 <> Ø
     end ;
14:  begin (*for1up*) h1 := s[t-1].i;
       if h1 <= s[t].i then s[s[t-2].i].i := h1 else
          begin t := t-3; pc := ir.y
          end
     end ;
15:  begin (*for2up*) h2 := s[t-2].i; h1 := s[h2].i + 1;
       if h1 <= s[t].i then
         begin s[h2].i := h1; pc := ir.y end
       else t := t-3;
     end ;
16:  begin (*for1down*) h1 := s[t-1].i;
       if h1 >= s[t].i then s[s[t-2].i].i := h1 else
          begin pc := ir.y; t := t-3
          end
     end ;
17:  begin (*for2down*) h2 := s[t-2].i; h1 := s[h2].i - 1;
       if h1 >= s[t].i then
         begin s[h2].i := h1; pc := ir.y end
       else t := t-3;
     end ;
18:  begin (*mark stack*)  h1 := btab[tab[ir.y].ref].vsize;
       if t+h1 > stacksize then ps := stkchk else
         begin t := t+5; s[t-1].i := h1-1; s[t].i := ir.y
         end
     end ;
19:  begin (*call*) h1 := t - ir.y;  (*h1 points to base*)
       h2 := s[h1+4].i;               (*h2 points to tab*)
       h3 := tab[h2].lev; display[h3+1] := h1;
       h4 := s[h1+3].i + h1;
       s[h1+1].i := pc; s[h1+2].i := display[h3]; s[h1+3].i := b;
       for h3 := t+1 to h4 do s[h3].i := Ø;
       b := h1; t := h4; pc := tab[h2].adr
     end ;
2Ø:  begin (*index1*) h1 := ir.y;      (*h1 points to atab*)
       h2 := atab[h1].low; h3 := s[t].i;
       if h3 < h2 then ps := inxchk else
       if h3 > atab[h1].high then ps := inxchk else
         begin t := t-1; s[t].i := s[t].i + (h3-h2)
         end
     end ;
21:  begin (*index*)  h1 := ir.y;      (*h1 points to atab*)
       h2 := atab[h1].low; h3 := s[t].i;
       if h3 < h2 then ps := inxchk else
       if h3 > atab[h1].high then ps := inxchk else
         begin t := t-1; s[t].i := s[t].i + (h3-h2)*atab[h1].elsize
```

```
          end
      end ;
  22: begin (*load block*) h1 := s[t].i; t := t-1;
        h2 := ir.y + t; if h2 > stacksize then ps := stkchk else
        while t < h2 do
          begin t := t+1; s[t] := s[h1]; h1 := h1+1
          end
      end ;
  23: begin (*copy block*) h1 := s[t-1].i;
        h2 := s[t].i; h3 := h1 + ir.y;
        while h1 < h3 do
          begin s[h1] := s[h2]; h1 := h1+1; h2 := h2+1
          end ;
        t := t-2
      end ;
  24: begin (*literal*) t := t+1;
        if t > stacksize then ps := stkchk else s[t].i := ir.y
      end ;
  25: begin (*load real*) t := t+1;
        if t>stacksize then ps := stkchk else s[t].r := rconst[ir.y]
      end ;
  26: begin (*float*) h1 := t - ir.y; s[h1].r := s[h1].i
      end ;
  27: begin (*read*)
        if eos(input) then ps := redchk else
          case ir.y of
           1: read(s[s[t].i].i);
           2: read(s[s[t].i].r);
           4: read(s[s[t].i].c);
          end ;
        t := t-1
      end ;
  28: begin (*write string*)
        h1 := s[t].i; h2 := ir.y; t := t-1;
        chrcnt := chrcnt+h1; if chrcnt > lineleng then ps := lngchk;
        repeat write(stab[h2]); h1 := h1-1; h2 := h2+1
        until h1 = Ø
      end ;
  29: begin (*write1*)
        chrcnt := chrcnt + fld[ir.y];
        if chrcnt > lineleng then ps := lngchk else
        case ir.y of
         1: write(s[t].i: fld[1]);
         2: write(s[t].r: fld[2]);
         3: write(s[t].b: fld[3]);
         4: write(s[t].c);
        end ;
        t := t-1
      end ;
  3Ø: begin (*write2*)
        chrcnt := chrcnt + s[t].i;
        if chrcnt > lineleng then ps := lngchk else
        case ir.y of
```

```
         1: write(s[t-1].i: s[t].i);
         2: write(s[t-1].r: s[t].i);
         3: write(s[t-1].b: s[t].i);
         4: write(s[t-1].c: s[t].i);
        end ;
        t := t-2
      end ;
31: ps := fin;
32: begin (*exit procedure*)
        t := b-1; pc := s[b+1].i; b := s[b+3].i
      end ;
33: begin (*exit function*)
        t := b; pc := s[b+1].i; b := s[b+3].i
      end ;
34: s[t] := s[s[t].i];
35: s[t].b := not s[t].b;
36: s[t].i := - s[t].i;
37: begin chrcnt := chrcnt + s[t-1].i;
        if chrcnt > lineleng then ps := lngchk else
           write(s[t-2].r: s[t-1].i: s[t].i);
        t := t-3
      end ;
38: begin (*store*) s[s[t-1].i] := s[t]; t := t-2
      end ;
39: begin t := t-1; s[t].b := s[t].r = s[t+1].r
      end ;
40: begin t := t-1; s[t].b := s[t].r <> s[t+1].r
      end ;
41: begin t := t-1; s[t].b := s[t].r < s[t+1].r
      end ;
42: begin t := t-1; s[t].b := s[t].r <= s[t+1].r
      end ;
43: begin t := t-1; s[t].b := s[t].r > s[t+1].r
      end ;
44: begin t := t-1; s[t].b := s[t].r >= s[t+1].r
      end ;
45: begin t := t-1; s[t].b := s[t].i = s[t+1].i
      end ;
46: begin t := t-1; s[t].b := s[t].i <> s[t+1].i
      end ;
47: begin t := t-1; s[t].b := s[t].i < s[t+1].i
      end ;
48: begin t := t-1; s[t].b := s[t].i <= s[t+1].i
      end ;
49: begin t := t-1; s[t].b := s[t].i > s[t+1].i
      end ;
50: begin t := t-1; s[t].b := s[t].i >= s[t+1].i
      end ;
51: begin t := t-1; s[t].b := s[t].b or s[t+1].b
      end ;
52: begin t := t-1; s[t].i := s[t].i + s[t+1].i
      end ;
53: begin t := t-1; s[t].i := s[t].i - s[t+1].i
```

```
     end ;
 54: begin t := t-1; s[t].r := s[t].r + s[t+1].r;
     end ;
 55: begin t := t-1; s[t].r := s[t].r - s[t+1].r;
     end ;
 56: begin t := t-1; s[t].b := s[t].b and s[t+1].b
     end ;
 57: begin t := t-1; s[t].i := s[t].i * s[t+1].i
     end ;
 58: begin t := t-1;
       if s[t+1].i = Ø then ps := divchk else
         s[t].i := s[t].i div s[t+1].i
     end ;
 59: begin t := t-1;
       if s[t+1].i = Ø then ps := divchk else
         s[t].i := s[t].i mod s[t+1].i
     end ;
 6Ø: begin t := t-1; s[t].r := s[t].r * s[t+1].r;
     end ;
 61: begin t := t-1; s[t].r := s[t].r / s[t+1].r;
     end ;
 62: if eos(input) then ps := redchk else readln;
 63: begin writeln; lncnt := lncnt + 1; chrcnt := Ø;
        if lncnt > linelimit then ps := linchk
     end
    end (*case*) ;
  until ps <> run;

  if ps <> fin then
  begin writeln;
    write('Øhalt at', pc:5, ' because of ');
    case ps of
      caschk: writeln('undefined case');
      divchk: writeln('division by Ø');
      inxchk: writeln('invalid index');
      stkchk: writeln('storage overflow');
      linchk: writeln('too much output');
      lngchk: writeln('line too long');
      redchk: writeln('reading past end of file');
    end ;
    h1 := b; blkcnt := 1Ø;   (*post mortem dump*)
    repeat writeln; blkcnt := blkcnt - 1;
      if blkcnt = Ø then h1 := Ø; h2 := s[h1+4].i;
      if h1<>Ø then
        writeln(' ', tab[h2].name, ' called at', s[h1+1].i: 5);
      h2 := btab[tab[h2].ref].last;
      while h2 <> Ø do
      with tab[h2] do
      begin if obj = variable then
            if typ in stantyps then
            begin write('    ', name, ' = ');
              if normal then h3 := h1+adr else h3 := s[h1+adr].i;
              case typ of
```

```
                 ints:  writeln(s[h3].i);
                 reals: writeln(s[h3].r);
                 bools: writeln(s[h3].b);
                 chars: writeln(s[h3].c);
                end
              end ;
              h2 := link
         end ;
         h1 := s[h1+3].i
       until h1 < Ø;
     end ;
     writeln; writeln(ocnt, ' steps')
   end (*interpret*) ;


   begin writeln;   {main program}
      key[ 1] := 'and       '; key[ 2] := 'array     ';
      key[ 3] := 'begin     '; key[ 4] := 'case      ';
      key[ 5] := 'const     '; key[ 6] := 'div       ';
      key[ 7] := 'downto    '; key[ 8] := 'do        ';
      key[ 9] := 'else      '; key[1Ø] := 'end       ';
      key[11] := 'for       '; key[12] := 'function  ';
      key[13] := 'if        '; key[14] := 'mod       ';
      key[15] := 'not       '; key[16] := 'of        ';
      key[17] := 'or        '; key[18] := 'procedure ';
      key[19] := 'program   '; key[2Ø] := 'record    ';
      key[21] := 'repeat    '; key[22] := 'then      ';
      key[23] := 'to        '; key[24] := 'type      ';
      key[25] := 'until     '; key[26] := 'var       ';
      key[27] := 'while     ';
      ksy[ 1] := andsy;        ksy[ 2] := arraysy;
      ksy[ 3] := beginsy;      ksy[ 4] := casesy;
      ksy[ 5] := constsy;      ksy[ 6] := idiv;
      ksy[ 7] := downtosy;     ksy[ 8] := dosy;
      ksy[ 9] := elsesy;       ksy[1Ø] := endsy;
      ksy[11] := forsy;        ksy[12] := functionsy;
      ksy[13] := ifsy;         ksy[14] := imod;
      ksy[15] := notsy;        ksy[16] := ofsy;
      ksy[17] := orsy;         ksy[18] := proceduresy;
      ksy[19] := programsy;    ksy[2Ø] := recordsy;
      ksy[21] := repeatsy;     ksy[22] := thensy;
      ksy[23] := tosy;         ksy[24] := typesy;
      ksy[25] := untilsy;      ksy[26] := varsy;
      ksy[27] := whilesy;
      sps['+'] := plus;        sps['-'] := minus;
      sps['*'] := times;       sps['/'] := rdiv;
      sps['('] := lparent;     sps[')'] := rparent;
      sps['='] := eql;         sps[','] := comma;
      sps['['] := lbrack;      sps[']'] := rbrack;
      sps['#'] := neq;         sps['&'] := andsy;
      sps[';'] := semicolon;
     constbegsys := [plus,minus,intcon,realcon,charcon,ident];
     typebegsys := [ident,arraysy,recordsy];
```

```
blockbegsys := [constsy,typesy,varsy,proceduresy,
                functionsy,beginsy];
facbegsys := [intcon,realcon,charcon,ident,lparent,notsy];
statbegsys := [beginsy,ifsy,whilesy,repeatsy,forsy,casesy];
stantyps := [notyp,ints,reals,bools,chars];
lc := 0; ll := 0; cc := 0; ch := ' ';
errpos := 0; errs := []; insymbol;
t := -1; a := 0; b := 1; sx := 0; c2 := 0;
display[0] := 1;
iflag := false; oflag := false;
if sy <> programsy then error(3) else
begin insymbol;
  if sy <> ident then error(2) else
  begin progname := id; insymbol;
    if sy <> lparent then error(9) else
    repeat insymbol;
      if sy <> ident then error(2) else
      begin if id = 'input     ' then iflag := true else
            if id = 'output    ' then oflag := true else error(0);
        insymbol
      end
    until sy <> comma;
    if sy = rparent then insymbol else error(4);
    if not oflag then error(20)
  end
end ;
enter('          ', variable, notyp, 0);  (*sentinel*)
enter('false     ', konstant, bools, 0);
enter('true      ', konstant, bools, 1);
enter('real      ', type1, reals, 1);
enter('char      ', type1, chars, 1);
enter('boolean   ', type1, bools, 1);
enter('integer   ', type1, ints , 1);
enter('abs       ', funktion, reals,0);
enter('sqr       ', funktion, reals,2);
enter('odd       ', funktion, bools,4);
enter('chr       ', funktion, chars,5);
enter('ord       ', funktion, ints, 6);
enter('succ      ', funktion, chars,7);
enter('pred      ', funktion, chars,8);
enter('round     ', funktion, ints, 9);
enter('trunc     ', funktion, ints, 10);
enter('sin       ', funktion, reals, 11);
enter('cos       ', funktion, reals, 12);
enter('exp       ', funktion, reals, 13);
enter('ln        ', funktion, reals, 14);
enter('sqrt      ', funktion, reals, 15);
enter('arctan    ', funktion, reals, 16);
enter('eof       ', funktion, bools, 17);
enter('eoln      ', funktion, bools, 18);
enter('read      ', prozedure, notyp, 1);
enter('readln    ', prozedure, notyp, 2);
enter('write     ', prozedure, notyp, 3);
```

```
  enter('writeln    ', prozedure, notyp, 4);
  enter('           ', prozedure, notyp, Ø);
  with btab[1] do
    begin last := t; lastpar := 1; psize := Ø; vsize := Ø
    end ;

  block(blockbegsys+statbegsys, false, 1);
  if sy <> period then error(22);
  emit(31);  (*halt*)
  if btab[2].vsize > stacksize then error(49);
  if progname = 'testØ     ' then printtables;

  if errs = [] then
  begin
    if iflag then
    begin getseg(input);
      if eof(input) then writeln(' input data missing') else
      begin writeln(' (eor)'); (*copy input data*)
        while not eos(input) do
        begin write(' ');
          while not eoln(input) do
            begin read(ch); write(ch)
            end ;
          writeln; read(ch)
        end ;
        getseg(input,Ø)
      end
    end ;
    writeln(' (eof)');
    interpret
  end
  else errormsg;
99:
end .
```

APPENDIX A: Syntax Diagrams

Notes

1. Round boxes denote symbols of the language Pascal, rectangular boxes denote syntactic constructs represented by diagrams.

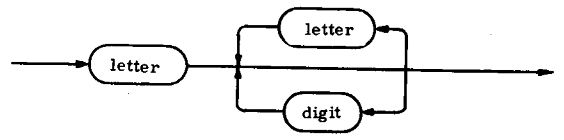
2. Separators may be inserted between any two symbols. However, no separators must occur within numbers and identifiers.

3. At least one separator must occur between consecutive identifiers, numbers, and word-symbols (such as BEGIN,END).

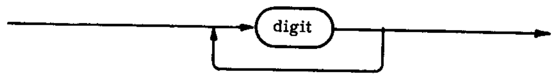
4. Separators are blanks, ends of lines, and comments. A comment is an arbitrary sequence of characters enclosed within a pair of comment brackets (* and *) .

5. The occurrence of the non-qualified word identifier in a syntax diagram implies that at this point an arbitrary, new identifier may be chosen. This identifier thereby becomes a constant-, type-, variable-, field-, function-, or procedure identifier.

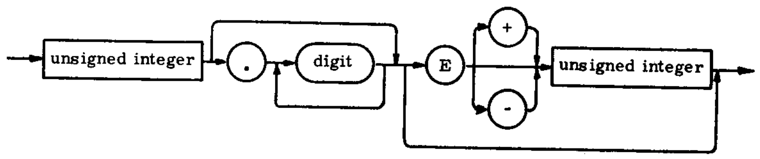
identifier



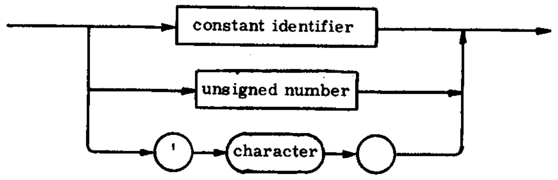
unsigned integer



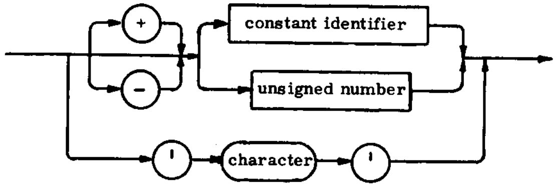
unsigned number



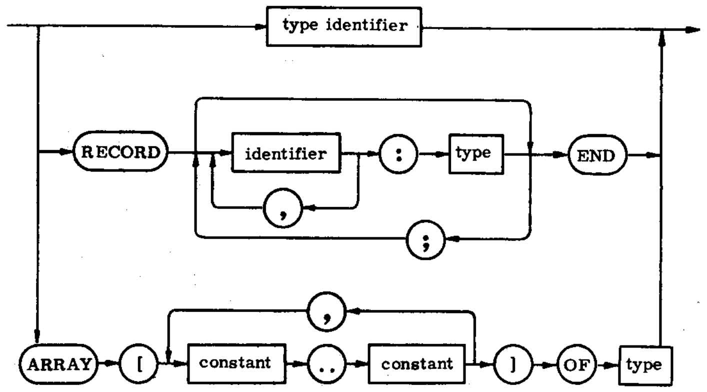
unsigned constant



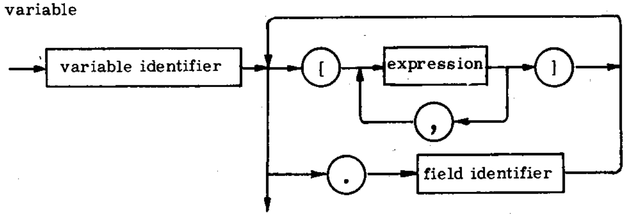
constant



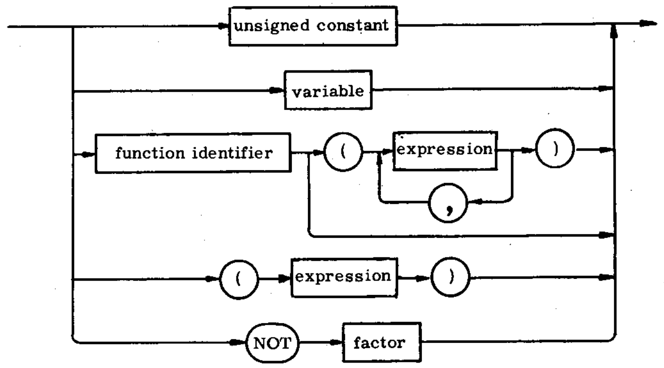
type



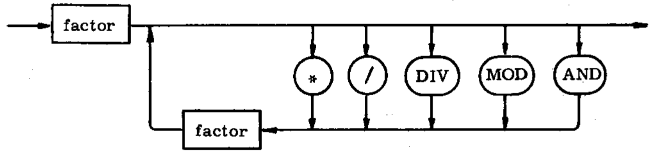
variable



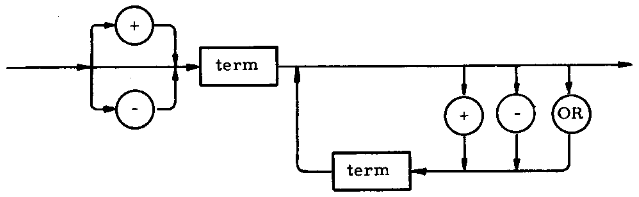
factor



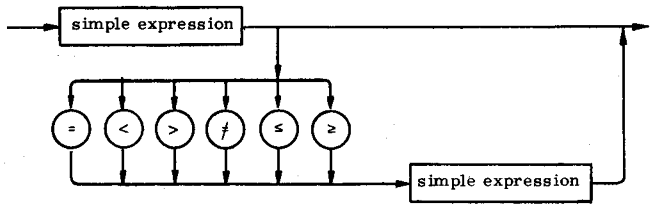
term



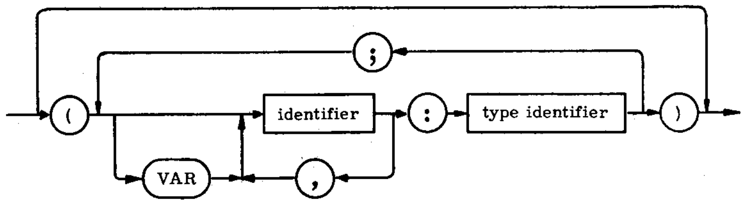
simple expression



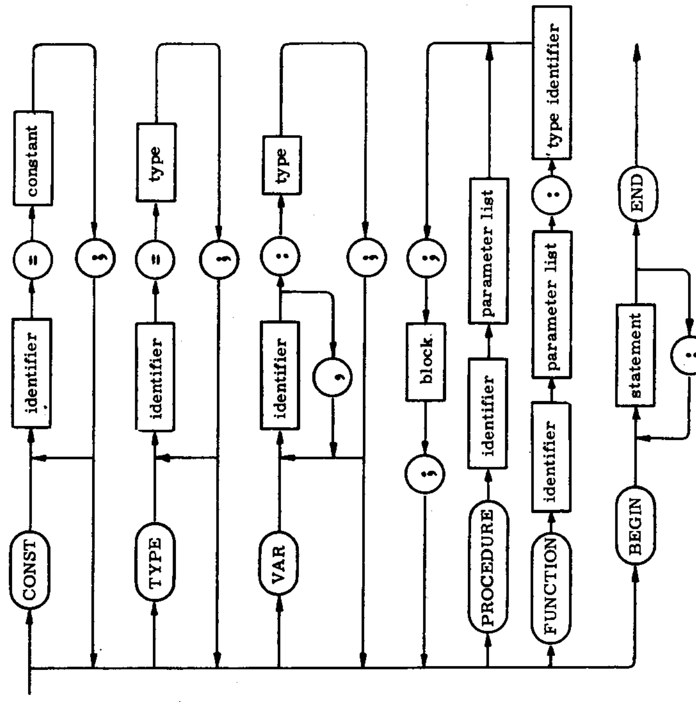
expression



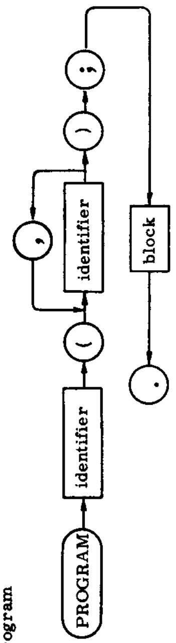
parameter list



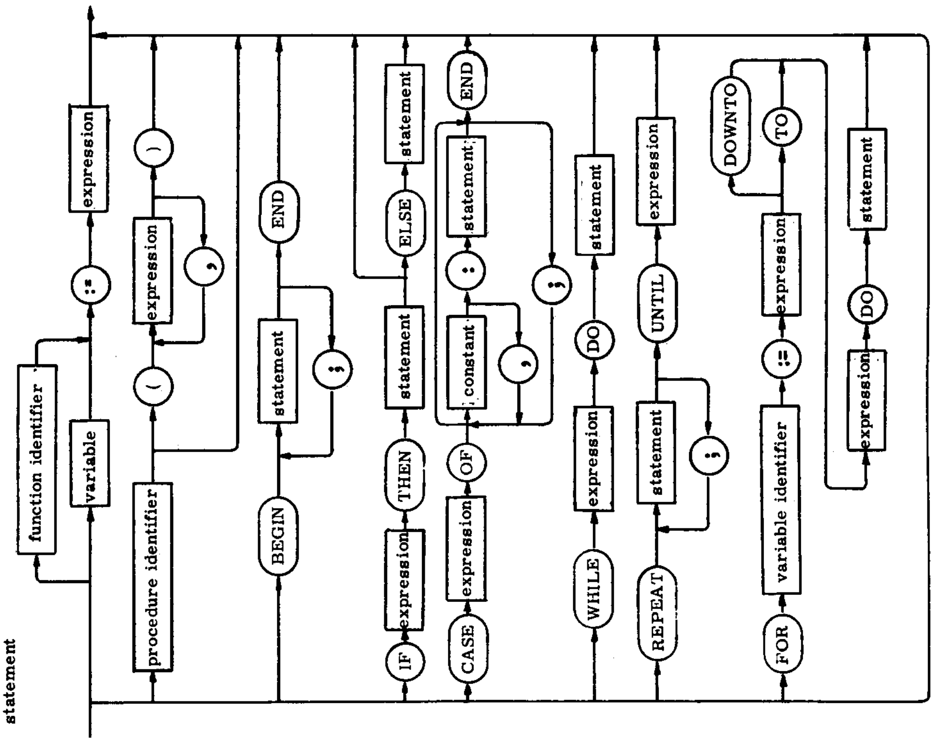
block



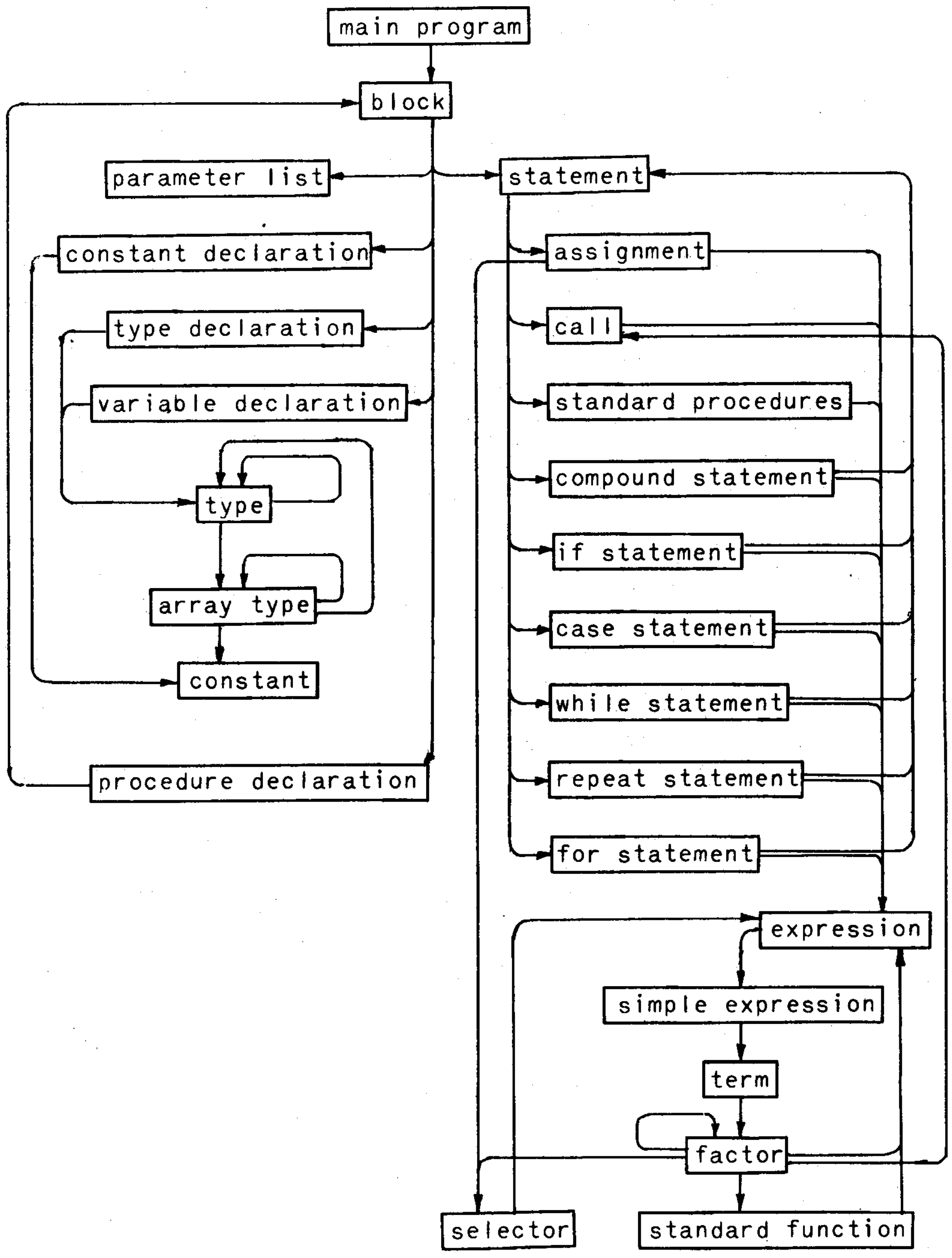
program



statement



APPENDIX B: Procedure Dependence Diagram



APPENDIX C: Explanations to Error Codes

Ø. The designated identifier has not been declared.
1. The indicated identifier is declared more than once in the same scope.
2. An identifier is expected.
3. Every program must begin with the symbol program.
4. A closing parenthesis is expected.
5. A colon is expected. In declarations, the colon is followed by a type.
6. At this point, the indicated symbol is incorrectly used. The compiler skips this and possibly several following symbols.
7. In a formal parameterlist each section must begin with an identifier or the symbol var, depending whether the parameter is a value or a variable parameter.
8. The symbol of is expected.
9. An opening parenthesis is expected.
1Ø. A type definition must begin with an identifier, the symbol array, or the symbol record.
11. An opening bracket is expected ([ ).
12. A closing bracket is expected (]).
13. The symbol .. is expected (no blank between the dots).
14. A semicolon is expected.
15. The result of a function must be of type integer, real, Boolean, or char.
16. An equal sign is expected. The symbol := is used in assignment statements only, but not in declarations.
17. The expression following the symbol if, while, or until must be of type Boolean.
18. The control variable following the symbol for must be of type integer, char, or Boolean.
19. The expressions which specify the initial and final values of the control variable in a for statement must be of the same type as the control variable.
2Ø. The parameter "output" must be included in the program heading.
21. The indicated number is too large. The maximum number of digits is 14; the absolute value must not exceed 1Ø**323 (on the CDC 6ØØØ implementation).
22. A dot is expected at the end of the program. Check corresponding begin and end symbols!
23. The expression following the symbol case must be of type integer, char, or Boolean. (In the latter case, an if statement is recommended.)
24. The designated character is not acceptable.
25. In a constant definition, the equal sign must be followed by a constant. If an identifier is used, it must denote a constant.
26. The type of an index expression must be identical to the index type specified in the array declaration.
27. In an array declaration, the lower bound must not exceed the upper bound. They must be within a permissible range of

values (less than 2**17). Also, their types must be identical, either integers, logical values, or characters. Real numbers are not acceptable.

28. Every indexed variable must be declared as an array.
29. A type identifier is expected here.
30. This type is not defined. (Recursive type definitions are not allowed.)
31. Every variable with a field selector must be declared as a record.
32. The operands of the operators not, and, and or must be of type Boolean.
33. The specified type of this arithmetic expression is illegal. Note also that entire arrays cannot occur as operands to arithmetic or logical operators.
34. Operands of div and mod must be of type integer.
35. The types of the comparands are incompatible. They must be identical, except if one comparand is of type integer and the other of type real. Arrays must be compared element by element.
36. The types of corresponding actual and formal parameters must be identical. An exception is made if the formal parameter is a value parameter of type real. Then the actual parameter may also be of type integer.
37. A variable is expected.
38. A string must contain at least one character.
39. The number of actual parameters must be equal to the number of specified formal parameters. 40. The parameters of the procedure read must be of type char, integer, or real.
41. The parameters of the procedure write must be of type char, integer, real, or Boolean.
42. If a statement has the form write(x:m:n), then x must be an expression of type real.
43. If a statement has the form write(x:n) or write(x:m:n), then m and n must be expressions of type integer.
44. No type or procedure identifiers may occur as part of an expression.
45. A statement cannot begin with a type or a function identifier. An exception is the assignment of a result value to a function. In this case, it must be part of the function body.
46. In an assignment x := y , the types of the variable x and the expression y must be identical. An exception is the case when x is real. Then, y may also be of type integer.
47. Every case label must be a constant of the same type as the expression in the case clause.
48. The indicated argument of the standard function is of an illegal type.
49. The program requires too much storage.
50. A constant cannot begin with the indicated symbol.
51. The symbol := is expected. (No space between : and =)
52. The symbol then is expected.
53. The symbol until is expected.
54. The symbol do is expected.

55. The symbol to (or downto) is expected,
56. The symbol begin is expected.
57. The symbol end is expected.
58. A factor must begin with an identifier, a constant, the symbol not, or with a left parenthesis.

References

1. U. Ammann, "The method of structured programming applied to the development of a compiler", International Computing Symposium 1973, 93-99, A. Guenther et al. (eds.), North-Holland (1974).

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

3. N. Wirth, "The programming language Pascal", Acta Informatica, Vol. 1, 35-63 (1971).

4. --------, "Systematisches Programmieren", Teubner-Verlag, Stuttgart, 1972.

5. --------, "Systematic Programming", Prentice-Hall, Inc., Englewood Cliffs, 1973.

6. --------, "Algorithms + Data Structures = Programs", Prentice-Hall, Inc., Englewood-Cliffs, Nov. 1975.