MOTOROLA

M68PLM(D1)

MOTOROLA PASCAL LANGUAGE MANUAL

SYSTEMS

MICROSYSTEMS

M68PLM(D1)

MOTOROLA

PASCAL

LANGUAGE MANUAL

Printed in Switzerland

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

PASCAL BACKGROUND

1.1 EARLY DEVELOPMENT

A preliminary version of the programming language Pascal was drafted by Prof. Niklaus Wirth at the Eidgenossiche Technische Hochschule, in Zurich, Switzerland, in 1968. It was based on the Algol-60 and Algol-W line of languages. Wirth's first compiler was operational in 1970.

With growing interest in the use of compilers for other computers and to incorporate some revisions to the language, a revised "Report" was published in 1973. This Motorola specification is based on the second edition* to that "Report". Material is used with permission of the Springer-Verlag Publishers, New York.

1.2 UCSD CONTRIBUTION

The Institute for Information Systems at the University of California at San Diego (UCSD), under the directorship of Kenneth Bowles, has been a major force behind the development of Pascal in the U.S. In 1974, Bowles looked to Pascal for its structured programming benefits in teaching; but he also wanted a language that would be portable and not locked into one particular computer. Through frequency-based encoding, he got the compiler working on the PDP-11 and, within a short time, on a microprocessor. Thereafter, he and his colleagues put together a single-user operating system that has received wide distribution and usage.

Once opened, the Pascal door swung wide for other compilers, which appeared with a variety of individualized extensions. The desirability of standardizing on these extensions was recognized by Pascal advocates and, in the summer of 1978, a workshop was hosted by the UCSD Institute for Information Systems. Out of this workshop arose a small set of extensions which sustained widespread industry support. The extensions agreed upon exhibited the following characteristics:

1. Compatibility - they did not invalidate programs written in the original language.
2. Convenience and Necessity - an extension was characterized either by being absolutely necessary for the intended application area, or by being highly convenient as compared to the unextended language.
3. Implementability - an extension was recommended only if an implementation could be proposed which did not lead to excessive translator complexity, or undue runtime or code space penalties.

Currently, the American National Standards Institute (ANSI) and the Institute of Electrical and Electronics Engineers (IEEE) have agreed to jointly develop a standard for the Pascal programming language.

* Kathleen Jensen and Niklaus Wirth, Pascal User Manual and Report, Second Edition (Springer-Verlag, New York, Heidelberg, Berlin 1974)

1.3 MOTOROLA FEATURES

Motorola's Pascal is based on the language as defined by Niklaus Wirth, with additions stimulated by Motorola's participation in the UCSD workshop and its current participation in the IEEE/ANSI standardization effort.

The first release of Pascal includes extensions for expressing certain embedded-control type operations, an important consideration to a large class of microprocessor users. Other extensions are desirable to users who will implement business-oriented systems. These extensions are as follows:

- address specification for variables
- alphanumeric labels
- exit statement
- external procedure and function declarations
- nondecimal integers
- otherwise clause in case statement
- relaxation of definition and declaration order
- runtime error checking
- runtime file assignment
- string operations
- string types
- underscores in identifiers

Omitted from the first release are formal procedure and function parameter specifications, packed structures, and type real. These standard Pascal features will be included in future releases.

Future releases of the language will include, as well, the following extensions:

- adjustable formal array dimensions
- compile-time initialization
- constant expressions
- half-length and double-length integers
- indexed files
- interrupt handling
- structured constants
- structured function values
- symbolic scalar I/O
- type transfer functions
- user abort

CHAPTER 2

BASIC LANGUAGE ELEMENTS

2.1 BASIC SYMBOLS

The Pascal vocabulary consists of the following basic symbols:

Letters : A through Z, a through z, and underscore (_)

Digits: Ø 1 2 3 4 5 6 7 8 9

Special Symbols: + - * / = < > ( ) [ ] . , ; : ' @ { } ↑

The following operators and delimiters, having a fixed meaning in the language, are formed using the above special symbols:

<> <= >= :=

(. and .)	Substitutes for [and] for delimiting array indices and sets.
(* and *)	Substitutes for { and } for delimiting comments.

The symbol @ is a substitute for ↑ for pointer types.

2.2 RESERVED WORDS

Reserved words are those integral parts of the Pascal language which a programmer cannot redefine. They are as follows:

and	exit	nil	set
array	file	not	string
begin	for	of	subprogram
case	forward	or	then
const	function	origin	to
div	goto	otherwise	type
do	if	procedure	until
downto	in	program	var
else	label	record	while
end	mod	repeat	with

2.3 SEPARATORS

Blanks, ends of lines, and comments are considered as separators. An arbitrary number of separators may occur between any two consecutive Pascal symbols, with the following restriction: no separators may occur within identifiers, numbers, and word symbols.

At least one separator must occur between any pair of consecutive identifiers, numbers, or word symbols.

CHAPTER 3

USER-SPECIFIED LANGUAGE ELEMENTS

3.1 IDENTIFIERS <identifier>*

Identifiers serve to denote constants, types, variables, procedures, and functions. Their association must be unique within their scope of validity -- i.e., within the procedure or function in which they are declared.

They consist of a letter or underscore followed by any combination of letters, digits, or underscores. The compiler does not distinguish between upper and lower case of letters, so these may be used at will to improve readability.

Identifiers denoting distinct objects must differ over the characters contained in the first ten positions.

Identifiers may not be reserved words (par. 2.2).

3.2 NUMBERS <unsigned number>

Numbers are constants of the predefined type integer. The integer range is -32768 through 32767.

Numbers can be written in the customary decimal notation or in other number bases where desired. The symbol # separates the base and number: the base written in decimal, and the number itself written in the base.

```
Examples:     1ØØ      (* 1ØØ is a number to the base 1Ø *)
              16#49FØ  (* 49FØ is a number to the base 16 *)
              8#377Ø   (* 377Ø is a number to the base 8 *)
```

3.3 STRINGS <string>

Strings are sequences of characters enclosed by apostrophes. A string consisting of a single character is a constant of the standard type char; a string consisting of n (>1) enclosed characters is a constant of the type <u>array</u> [1..n] <u>of</u> char, and is compatible with <u>string</u> type.

If the string is to contain an apostrophe, then the apostrophe is to be written twice.

```
Examples:     'A'
              'don''t'
              'this is a string'
```

* A syntactic apex variable which equates to an entry in Appendix A, "Syntax". A similar variable is found at the top of each Pascal segment in the manual, directing the user to the fully expanded syntax.

3.4 COMMENTS

A comment is a sequence of characters enclosed by the symbol pairs (* and *). The symbol pairs (* and *) are used as synonyms for { and }.

A comment may be inserted between any two identifiers, numbers, or special symbols, and it may be replaced by a space without altering the meaning of the program text.

However, a "comment" enclosed by apostrophes is not a comment, but a string. For example, '(* program name *)' is a string.

```
Examples:    (* beginning of program *)
             (* send message to printer *)
```

CHAPTER 4

PROGRAMS

4.1 INTRODUCTION <program>

A Pascal program has the form of a procedure declaration (chapter 5) except for its heading.

4.2 PROGRAM HEADING <program heading>

A program heading contains, in the following order: (1) the symbol program, (2) a program name identifier, and (3) program parameter identifier(s).

The identifier which follows the symbol program is the program name; it has no further significance inside the program.

The two standard files, input and output, are listed as parameters in the program heading, if they are used. They are predeclared as

```
     var  input, output: text
```

and should not be declared in the variable declaration part (par. 5.3.2).

4.3 DECLARATION PART <declaration part>

The declaration part contains declarations and definitions which are local to the program. For details, see paragraph 5.3.

4.4 STATEMENT PART <statement part>

The statement part of a program has the structure of a single compound statement (par. 7.2.1). The word end, however, that terminates the top-level compound statement is followed by a period (.), which terminates the program.

Example:

```
program copytext(input,output);
var ch: char;
begin
   while not eof(input) do
   begin
      while not eoln(input) do
         begin read(ch); write(ch)
         end;
      readln; writeln
   end
end.
```

CHAPTER 5

PROCEDURE DECLARATION

5.1 INTRODUCTION <procedure declaration>

The procedure declaration serves to define a program part and to associate an identifier with it so that it can be activated by a procedure statement (par. 7.1.2). The declaration consists of a "procedure heading", a "declaration part", and a "statement part".

If the procedure is defined later in the compilation or externally in a separate compilation, the directive forward replaces the declaration and statement parts. Forward, when used, follows the procedure heading.

Example:

```
procedure forml(k,l: integer); forward;
```

5.2 PROCEDURE HEADING <procedure heading>

The procedure heading specifies the identifier naming the procedure, and an optional formal parameter section(s).

The parameters are either value-, variable-, function-, or procedure parameters. A parameter group without preceding specifier (i.e., var, function, or procedure) implies that its constituents are value parameters.

Examples:

```
procedure sum (var i,j: integer); forward;


(* Assign to x the value of the next integer in the textfile f *)
procedure readinteger (var f: text; var x: integer);
var i,j: integer;
begin while f@ = ' ' do get(f); i:=Ø;
   while f@ in ['Ø'..'9'] do
      begin j:= ord(f@) - ord('Ø');
         i:= 1Ø * i + j;
         get(f)
      end;
   x:= i
end
```

5.3 DECLARATION PART <declaration part>

The declaration part comprises the declaration of labels, variables, procedures, and functions, and the definition of constants and types.

5.3.1 Label Declaration Part <label declaration part>

The label declaration part, which is introduced by the symbol label, specifies all labels which mark statements in the statement part (pars. 7.1.3 and 7.1.5). Labels are either identifiers (par. 3.1), or unsigned decimal numbers of four digits or less.

Example:
```
    (* 2001 and checkit are labels in the current procedure *)
    label 2001, checkit
```

5.3.2 Variable Declaration Part <variable declaration part>

The variable declaration part, which is introduced by the symbol var, contains all variable declarations local to the procedure declaration. A variable is declared by denoting its identifier, followed optionally by the memory address (origin) at which it is to reside, followed by its previously defined type.

Variables may be entire, components of an array, record or file, or pointers.

Example:
```
    var i,j: integer; (* Declare i and j to be integer variables *)

        (* Declare a 10-element array of records, each containing an
           integer field and a character field *)
        arr: array [1..10] of
           record
              f1: integer;
              f2: char
           end;

        x: @integer; (* Declare x to be a pointer to an integer *)

        a [origin 16#FCF4]: ACIA; (* Declare a to be an ACIA at hex
                                     memory address FCF4 *)
```

5.3.2.1 Entire Variables. <entire variable> An entire variable is denoted by its identifier.

5.3.2.2 Component Variables. <component variable> A component of a structured variable is denoted by the variable identifier, followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

5.3.2.2.1 Indexed Variables. <indexed variable> A component of an n-dimensional array variable is denoted by the variable, followed by n index expressions. The types of the index expressions must correspond with the index types declared in the definition of the array type.

Examples:
```
    a [12]
    a [i + j]
    b [red,true]
```

5.3.2.2.2 Field Designators. <field designator> A component of a record variable is denoted by the record variable, followed by the field identifier of the component.

Examples:
```
u.re
b [red,true].im
p2@.size
```

5.3.2.2.3 File Buffers. <file buffer> At any time, only the one component determined by the current position of a file "read/write head" is directly accessible. This component is called the current file component and is represented by the file's buffer variable.

Example:
```
f@
```

5.3.2.3 Referenced Variables. <referenced variable> If p is a pointer variable which is bound to a type T, p denotes that variable and its pointer value, whereas p@ denotes the variable of type T referenced by p.

Examples:
```
p1@
p1@.father
p1@.sibling@.child
```

5.3.3 Procedure Declaration Part <procedure declaration>

The procedure declaration, which is introduced by the symbol procedure, serves to define a program part (par. 5.1). If procedures are nested, this procedure declaration part defines those nested program parts.

5.3.4 Function Declaration Part <function declaration>

The function declaration part, which is introduced by the symbol function, serves to define a program part which computes a value (par. 8.1).

5.3.5 Constant Definition Part <constant definition part>

The constant definition part, which is introduced by the symbol const, contains all constant synonym definitions local to the procedure. A constant definition introduces an identifier as a synonym for a constant.

The constants may be signed/unsigned numbers, signed/unsigned constant identifiers, or strings.

Examples:
```
const maxindex = Ø;
      rev = 'Ø3.ØØ';
```

5.3.6 Type Definition Part <type definition part>

The type definition part, which is introduced by the symbol type, contains all data type definitions which are local to the procedure declaration. A data type determines the set of values which variables of that type may assume, and associates an identifier with the type. Table 5-1 lists the various data types.

Example:
```
type T = array [1..1Ø] of string [1Ø];
```

TABLE 5-1. Data Types

Simple	Structured	Pointer
Scalar	Array	Pointer
Subrange	Record	
Standard simple	Set	
	File	
	String	

5.3.6.1 Scalar Types. <scalar type> A scalar type defines an ordered set of values by enumeration of the identifiers which denote these values.

Examples: type colors = (red, orange, yellow, green, blue);

suits = (club, diamond, heart, spade);

days = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

5.3.6.2 Subrange Types. <subrange type> A type may be defined as a subrange of another scalar type by indication of the least and the greatest value in the subrange. The first constant specifies the lower bound, and must not be greater than the upper bound.

Examples: type index = 1..1ØØ;

range = -1Ø..+1Ø;

weekdays = Monday..Friday;

5.3.6.3 Standard Simple Types. The following type identifiers are standard in Pascal: integer, Boolean, and char. However, other type identifiers may be defined by the programmer.

integer	The values are a subset of the whole numbers defined by individual implementations. Its values are denoted by integers (par. 3.2).
Boolean	Its values are the truth values denoted by the predefined identifiers true and false, such that false is less than true.
char	Its values are a set of characters determined by particular implementations. They are denoted by the characters themselves enclosed within apostrophes.

5.3.6.4 Array Types. <array type> An array type is a structure consisting of a fixed number of components which are all of the same type, called the component type. The elements of the array are designated by indices, values belonging to the so-called index type. The array type definition specifies the component type as well as the index type.

An array can have n >= Ø (i.e., unlimited) dimensions. When n index types are specified, the array type is called n-dimensional, and a component is designated by n indices.

Examples: type array1 = array [1..1ØØ] of integer;

array2 = array [1..1Ø,1..2Ø] of Ø..99;

array3 = array [Boolean] of color;

5.3.6.5 Record Types. <record type> A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a field, its type and an identifier which denotes it. The scope of these so-called field identifiers is the record definition itself, and they are also accessible within a field designator referring to a record variable of this type.

A record type may have several variants, in which case a certain field may be designated as the tag field, whose value indicates which variant is assumed by the record variable at a given time. Each variant structure is identified by a case label which is a constant of the type of the tag field.

When the tag field ("s:" in the example below) is present, a runtime error will occur on accessing any field in a variant other than the one corresponding to the current value of the tag field. The tag field may be omitted, in which case no variant error checking is performed.

```
Examples:    type date = record day: 1..31;
                               month: 1..12;
                               year: integer
                        end;

                  person = record name, firstname: alfa;
                                  age: Ø..12Ø;
                                  married: Boolean
                           end;

                  object = record x,y: integer;
                                  A: area;
                                  case s: shape of
                                  triangle: (side: integer;
                                            inclination, angle1, angle2: angle);
                                  rectangle: (side1, side2: integer;
                                             skew, angle3: angle);
                                  circle: (diameter: integer)
                           end;
```

5.3.6.6 Set Types. <set type> A set type defines the range of values which is the powerset of its so-called base type. Base types must not be structured types.

```
Examples:    type digits = set of Ø..9;
                  patriotic = set of (red,white,blue);
```

5.3.6.7 File Types. <file type> A file type definition specifies a structure consisting of a sequence of components which are all of the same type. The number of components, called the length of the file, is not fixed by the file type definition. A file with zero components is called empty.

Components of a file are accessed sequentially. Operations for stepping through a file while reading or writing, and for resetting to the beginning, are described in Chapter 9.

Sequential files with component type char are called textfiles, and are a special case insofar as the range of component values may be considered extended by a marker denoting the end of a line. This marker allows textfiles to be substructured into lines. The type text is a standard type, predeclared as

```
            type text = file of char
```

Example: type report = file of char;

5.3.6.8 String Types. <string type> An object of string type can take on values of sequences of characters. The sequences have lengths from one character up to a number defined by the context. Specifically, a particular string type is specified with a maximum length that objects of that type may possess.

Example: type name = string [1Ø]

5.3.6.9 Pointer Types. <pointer type> Variables which are declared in a program are accessible by their identifier. They exist during the entire execution process of the procedure (scope) to which the variable is local, and these variables are, therefore, called static (or statically allocated). In contrast, variables may also be generated dynamically -- i.e., without any correlation to the structure of the program. These dynamic variables are generated by the standard procedure new; since they do not occur in an explicit variable declaration, they cannot be referred to by a variable name. Instead, access is achieved via a so-called pointer value which is provided upon generation of the dynamic variable. A pointer type thus consists of an unbounded set of values pointing to elements of the same type.

```
Examples:     type link = @integer;
                   ptype = @person;
```

5.4 STATEMENT PART <statement part>

The statement part takes the form of a compound statement (par. 7.2.1).

5.5 STANDARD PROCEDURES

Standard procedures are predeclared in every implementation of Pascal. Motorola implementations feature additional predeclared procedures, which are denoted in this manual with an asterisk (*). Since they are assumed as declared in a scope surrounding the program, as are all standard quantities, no conflict arises from a declaration redefining the same identifier within the program. The standard procedures are listed and explained below.

5.5.1 File Handling Procedures

The standard file handling procedures are described in Chapter 9.

5.5.2 Dynamic Allocation Procedures

new(p)	Allocates a new variable v and assigns the pointer to v to the pointer variable p. If the type of v is a record type with variants, the form
new(p, t1,...,tn)	Can be used to allocate a variable of the variant with tag field values t1,...,tn. The tag field values must be listed contiguously and in the order of their declaration and must not be changed during execution.
dispose(p)	Indicates that storage occupied by the variable p@ is no longer needed. If the second form of new was used to allocate the variable then
dispose(p, t1,...,tn)	With identical field values must be used to indicate that storage occupied by this variant is no longer needed.

5.5.3 String Procedures

*delete(s,x,y)	Delete a substring from the given string (s) beginning at the indicated position (x) and running for the given number of characters (y).
*insert(s1,s2,x)	Insert string 1 (s1) at the given position (x) of string 2 (s2). Move any trailing characters in string 2 to the right, past the insertion. If necessary, truncate the resulting string so that it conforms to the maximum size of string 2.

CHAPTER 6

EXPRESSIONS

6.1 INTRODUCTION <factor>,<term>,<simple expression>,<expression>

Expressions are constructs denoting rules of computation for obtaining values. Their uses include generating new values for variables by the application of operators. Expressions consist of operators and operands -- i.e., variables, constants, and functions.

Expressions which are members of a set must all be of the same type, which is the base type of the set. [] denotes the empty set, and [x..y] denotes the set of all values in the interval x...y.

Examples:

```
Factors:               x
                       15
                   (x + y + z)
                  [red, c, green]
                 [1, 5, 10..19, 23]
                     not p

Terms:               x * y
                    i/(1 - i)
                (x <= y) and (y > z)

Simple
  expressions:       x + y
                      - x
                     p or q
                   hue1 + hue2
                    i * j + 1

Expressions:         p <= q
                  (i < j) = (j < k)
                    c in hue1
```

6.2 OPERATORS

The rules of composition specify operator precedences according to four classes of operators. The not operator has the highest precedence, followed by the multiplying operators, then the adding operators and, finally, with the lowest precedence, the relational operators (see Tables 6-1 through 6-4). Sequences of operators of the same precedence are executed from left to right.

Notice that all scalar types define ordered sets of values.

The operators <>, <=, >= stand for unequal, less or equal, and greater or equal, respectively.

The operators <= and >= may also be used for comparing values of set type, and then denote set inclusion.

If p and q are Boolean expressions, p = q denotes their equivalence, and p <= q denotes implication of q by p. (Note that false <= true).

The relational operators = <> < <= > >= may also be used to compare arrays having components of type char and to compare strings. They then denote alphanumeric ordering according to the binary collating sequence of characters in the ASCII character set. For equal length strings, characters in corresponding character positions are compared, starting from the high-order position, until either a pair of unequal characters or the low-order position of the string is compared. For unequal length strings, the comparison proceeds as for strings of equal length. If the process exhausts the characters of the shorter string, the shorter string is less than the longer unless the remainder of the longer string consists solely of ASCII spaces, in which case the strings are equal.

When used as operators with one operand only, - denotes sign inversion, and + denotes the identity operation.

TABLE 6-1. Not Operator

Operator	Operation	Type of Operands	Type of Result
not	Negation	Boolean	Boolean

TABLE 6-2. Multiplying Operators

Operator	Operation	Type of Operands	Type of Result
*	Multiplication	Integer	Integer
	Set Intersection	Any Set Type T	T
div	Division with Truncation	Integer	Integer
mod	Modulus	Integer	Integer
and	Logical "and"	Boolean	Boolean

TABLE 6-3. Adding Operators

Operator	Operation	Type of Operands	Type of Result
+	Addition	Integer	Integer
	Set Union	Any Set Type T	T
-	Subtraction	Integer	Integer
	Set Difference	Any Set Type T	T
or	Logical "or"	Boolean	Boolean

TABLE 6-4. Relational Operators

Operator	Operation	Type of Operands	Type of Result
=	"equal to"	Any Scalar, Subrange, or Set Type	Boolean
<>	"not equal to"	(Same as above)	Boolean
<	"less than"	Any Scalar or Subrange Type	Boolean
>	"greater than"	(Same as above)	Boolean
<=	"less than or equal to", implication, set inclusion	Any Scalar, Subrange, or Set Type	Boolean
>=	"greater than or equal to", set containment	(Same as above)	Boolean
in	Set membership	Any Scalar or Subrange Type & Its Set Type Respectively	Boolean

6.3 FUNCTION DESIGNATORS <function designator>

A function designator specifies the activation of a function. It consists of the identifier designating the function and a list of actual parameters. The parameters are variables, expressions, procedures, and functions, and are substituted for the corresponding formal parameters.

Examples:

```
Sum(a, 100)
GCD(147, k)
eof(f)
ord(f@)
```

CHAPTER 7

STATEMENTS

Statements denote algorithmic actions, and are said to be executable. They may be prefixed with a label (par. 5.3.1) followed by a colon, which enables that statement to be referenced by goto and exit statements.

7.1 SIMPLE STATEMENTS <simple statement>

A simple statement is a statement of which no part constitutes another statement. In this group are the assignment, procedure, goto, empty, and exit statements.

7.1.1 Assignment Statements <assignment statement>

The assignment statement serves to replace the current value of a variable or a function identifier by a new value specified as an expression.

The variable (or the function) and the expression must be of identical type. One exception, however, is permitted -- i.e., the type of the expression is a subrange of the type of the variable, or vice-versa.

```
Example:    x := y + z  (* Replace current value of x by sum of y and z *)
```

7.1.2 Procedure Statements <procedure statement>

A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement may contain a list of actual parameters which are substituted in place of their corresponding formal parameters defined in the procedure declaration. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters, respectively. There exist four kinds of parameters: so-called value parameters, variable parameters, procedure parameters, and function parameters.

In the case of a value parameter, the actual parameter must be an expression (of which a variable is a simple case). The corresponding formal parameter represents a local variable of the called procedure, and the current value of the expression is initially assigned to this variable. In the case of a variable parameter, the actual parameter must be a variable, and the corresponding formal parameter represents this actual variable during the entire execution of the procedure. If this variable is a component of an array, its index is evaluated when the procedure is called. A variable parameter must be used whenever the parameter represents a result of the procedure.

```
Examples:   next;

            transpose(a,n,m);
```

7.1.3 Goto Statements <goto statement>

A goto statement serves to indicate that further processing should continue at another part of the program text -- namely, at the place of the label.

The following restrictions hold concerning the applicability of labels:

1. The scope of a label is the procedure within which it is defined. It is, therefore, not possible to jump into, or out of, a procedure.

2. Every label must be specified in a label declaration in the heading of the procedure in which the label marks a statement.

Example: _goto_ sampl;

7.1.4 Empty Statements <empty statement>

The empty statement consists of no symbols and denotes no action. It occurs whenever the syntax of Pascal requires a statement but no statement appears.

Example:
```
(* The statement encountered between begin and end is the empty
   statement which, in this case, has no effect *)

begin

end;
```

7.1.5 Exit Statements <exit statement>

The execution of an _exit_ statement without a label operand causes the immediate termination of the smallest enclosing repetitive statement. Control is given to the same statement that would be executed after normal loop termination. If a label is given, then control is given to the statement that would be executed after normal termination of the enclosing repetitive statement carrying the given label.

Examples:
```
       for i := 1 to 2Ø do begin
          read(file1, array1 [i]);
          if eoln(file1) then exit
       end;
         (* Control results here after both count-triggered and eoln-type
          terminations. *)

 lab2: for i := 1 to 2Ø do
          begin
             j := 2;
             repeat
                array1 [i] := array1 [i] + array2 [j];
                if array2 [j] = 999 then exit lab2;
                j := j + 2
             until j = 1ØØ
             (* control results here after Boolean expression becomes
                true *)
          end;
          (* control results here after count-triggered, or if the exit
             statement with label lab2 is executed *)
```

7.2 STRUCTURED STATEMENTS

Structured statements are constructs composed of other statements which have to be executed in sequence (compound statement), conditionally (conditional statements), repeatedly (repetitive statements), or by a with statement.

7.2.1 Compound Statements <compound statement>

The compound statement specifies that its component statements are to be executed in the same sequence in which they are written. The symbols begin and end act as statement brackets. The statements that make up the compound statement are separated by semicolons (;).

```
Example:  begin  z := x; x := y; y := z end; (* Interchange values of x and y,
                                                using z *)
```

7.2.2 With Statements <with statement>

Within the component statement of the with statement, the components (fields) of the record variable(s) specified in the record variable list can be denoted by their field identifier only -- i.e., without preceding them with the denotation of the entire record variable. The with clause effectively opens the scope containing the field identifiers of the specified record variable(s), so that the field identifiers may occur as variable identifiers.

If the variable date is declared as

```
          var  date: record
                       month: integer;
                       year: integer
                     end
```

then

```
          with date do
          if month = 12 then
             begin month := 1; year := year + 1
             end
          else month := month + 1
```

is equivalent to

```
          if date.month = 12 then
             begin date.month := 1; date.year := date.year + 1
             end
          else date.month := date.month + 1
```

If the selection of a variable in the record variable list involves the indexing of an array or the dereferencing of a pointer, then these actions are executed before the component statement is executed.

7.2.3 Conditional Statements <conditional statement>

A conditional statement selects for execution a single one of its component statements.

7.2.3.1 _If Statements_. <if statement> The _if_ statement specifies that a statement be executed only if a certain condition (Boolean expression) is true. If it is false, then either no statement is to be executed, or the statement following the symbol _else_ is to be executed.

The syntactic ambiguity arising from the construct

```
    if <expression1> then if <expression2> then <statement1> else <statement2>
```

is resolved by interpreting the construct as equivalent to

```
    if <expression1> then
    begin
       if <expression2> then <statement1> else <statement2>
    end
```

Examples:

```
if x < 15 then z := x + y else z := 15;
if p1 < > nil then p1 := p1@.father;
```

7.2.3.2 _Case Statements_. <case statement> The _case_ statement consists of an expression (the selector) and a list of statements, each being labeled by a constant of the type of the selector. It specifies that the one statement be executed whose label is equal to the current value of the selector. If no label equals the value of the selector, control is given to the statement in the _otherwise_ clause if it exists. Otherwise, the statement causes a runtime error.

Examples:

```
case operator of
  plus: x := x + y;
  minus: x := x - y;
  times: x := x * y
end;
case i of
  1: x := abs(x);
  2: x := sqr(x);
  3: x := succ(x);
  4: x := pred(x)
  otherwise: x := Ø
end;
```

7.2.4 Repetitive Statements <repetitive statement>

Repetitive statements specify that certain statements are to be executed repeatedly. If the number of repetitions is known beforehand -- i.e., before the repetitions are started -- the _for_ statement is the appropriate construct to express this situation; otherwise, the _while_ or _repeat_ statement should be used.

7.2.4.1 While Statements. <while statement> In the while statement, the component statement is repeatedly executed while the controlling Boolean expression is true. If the expression's value is false at the beginning, the component statement is not executed at all.

```
          while B do S
```

is equivalent to

```
          if B then
            begin S;
              while B do S
            end
```

Examples:

```
          while a[i]< > x do i := i + 1;

          while i > Ø do
            begin if odd(i) then z := z * x; i := i div 2; x := sqr(x)
            end;

          while not eof(f) do
            begin P(f@); get(f)
            end;
```

7.2.4.2 Repeat Statements. <repeat statement> The repeat statement provides for the repetition of a sequence of statements based on a controlling expression. The expression controlling repetition must be of type Boolean. The sequence of statements between the symbols repeat and until is repeatedly executed (and at least once) until the expression becomes true. The repeat statement

```
          repeat S until B
```

is equivalent to

```
          begin S;
            if not B then
              repeat S until B
          end
```

Examples:

```
          repeat k := i mod j; i := j; j:=k
          until j = Ø;

          repeat P(f@); get(f)
          until eof(f);
```

7.2.4.3 For Statements. <for statement> The for statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the control variable of the for statement. The progression can be up to or downto a final value.

The control variable, the initial value, and the final value must be of the same scalar type (or subrange thereof), and must not be altered by the repeated statement.

The for-statement

```
for v := el to e2 do body
```

is equivalent to

```
begin
templ := el; temp2 := e2;
if templ <= temp2 then
  begin
  v := templ;
  body;
  while v <> temp2 do
    begin
    v := succ(v);
    body
    end
  end
end
```

and the for-statement

```
for v := el downto e2 do body
```

is equivalent to

```
begin
templ := el; temp2 := e2;
if temp 1 >= temp2 then
  begin
  v := templ;
  body;
  while v <> temp2 do
    begin
    v := pred(v);
    body
    end
  end
end
```

where templ and temp2 are auxiliary variables of the host type of the variable v which do not occur elsewhere in the program.

Examples:

```
for i := 2 to 63 do if a[i] > max then max := a[i];

for i := 1 to n do
for j := 1 to n do
begin x := 0;
   for k := 1 to n do x := x + A[i, k] *B[k, j]; C[i, j] := x
end;

for c := red to blue do Q(c);
```

CHAPTER 8

FUNCTION DECLARATIONS

8.1 INTRODUCTION <function declaration>

The function declaration serves to define a program part that computes a value. The declaration consists of a "function heading", a "declaration part", and a "statement part".

If the function is defined later in the compilation or externally in a separate compilation, the directive forward replaces the declaration and statement parts. Forward, when used, follows the function heading.

Functions are activated by the evaluation of a function designator (par. 6.3), which is a constituent of an expression.

8.2 FUNCTION HEADING <function heading>

The function heading specifies the identifier naming the function, an optional formal parameter section(s), and the result type.

The parameters are either value-, variable-, function-, or procedure parameters. A parameter group without preceding specifier (i.e., var, function, or procedure) implies that its constituents are value parameters.

Example:

```
function Power (x,y: integer): integer;   (* y > = Ø *)
var w,z,i: integer;
begin w := x; z := 1;i := y;
   while i > Ø do
   begin (* z* (w**i) = x ** y *)
     if odd(i) then z := z*w;
     i := i div 2;
     w := sqr(w)
   end
   (* z = x**y *)
   Power := z
end;

function  GCD (m,n: integer): integer; forward;
```

8.3 DECLARATION PART <declaration part>

The declaration part comprises the declaration of labels, variables, procedures, and functions, and the definition of constants and types (par. 5.3).

8.4 STATEMENT PART <statement part>

The statement part takes the form of a compound statement (par. 7.2.1). There must be at least one assignment statement assigning a value to the function identifier. This assignment determines the result of the function.

Occurrence of the function identifier in a function designator within its declaration implies recursive execution of the function.

8.5 STANDARD FUNCTIONS

Standard functions in Table 8-1 are predeclared in every implementation of Pascal. Motorola Pascal, however, features additional predeclared functions, which are marked in the text with an asterisk.

TABLE 8-1. Standard Functions

FUNCTION	IDENTIFIER	MEANING
Arithmetic	abs(x)	The absolute value of x.
	sqr(x)	x raised to the power of 2.
Predicates	odd(x)	A Boolean result of true if x is odd; false otherwise. The type of x must be integer.
	eof(f)	Indicates whether the file f is in the end-of-file status. (chapter 9)
	eoln(f)	Indicates the end of a line in a textfile. (chapter 9)
Transfer	ord(x)	x must be of a scalar type (including Boolean and char), and the result (of type integer) is the ordinal number of the value x in the set defined by the type of x.
	chr(x)	x must be of type integer, and the result (of type char) is the character whose ordinal number is x (if it exists).
Enumera-tive	succ(x)	x is of any scalar or subrange type, and the result is the successor value of x (if it exists).
	pred(x)	x is of any scalar or subrange type, and the result is the predecessor value of x (if it exists).
*String	length(s)	Return the current length of the given string (s).
	pos(s1,s2)	Return the position of the first occurrence of string 2 (s2) in string 1 (s1). If there is no such occurrence, return zero.
	concat(s1, s2,...,sn)	Return a string equal to string 1 (s1) with string 2 (s2) through string n (sn) concatenated onto its end.
	copy(s,x,y)	Return the substring of the given string (s) that begins at the indicated position (x) and runs for the specified number of characters (y).

CHAPTER 9

INPUT AND OUTPUT

9.1 INTRODUCTION

Many Pascal programs will communicate with their environment solely or principally through Pascal's I/O facilities. The Pascal object through which this takes place is the file. Variables of file type represent ordered collections of components; files are differentiated as to type of data component -- in particular, the kind of file with component type char is called a file of type text and has special formatting facilities available.

9.2 CORRESPONDENCE OF FILE VARIABLES TO SYSTEM I/O RESOURCES

A file variable can correspond to a system resource which might be a storage, display, or transmission facility. The act of creating this correspondence is called assignment and can be accomplished in a variety of ways depending on requirements. The principal distinction concerning the type of assignment is between local and external files.

9.2.1 Local Files

Some file variables represent data with a useful lifetime that does not extend beyond their program's execution. These are commonly known as scratch, or local files. If a file variable is not assigned as external (par. 9.2.2), the system will cause a local file to be created when the block containing its definition is entered, and to be destroyed when that block is exited. If the block is the program's outer block, then the file's lifetime is effectively that of the program's execution.

9.2.2 External Files

Files which represent data or facilities that exist before and/or after the program's execution are called external; they have an existence outside the program and are known to the system by names that are not related to the file variable name. Therefore, the assignment process must establish the name-correspondence at file opening time.

This is accomplished by using a variation of reset and rewrite that takes a second argument, an expression of string type. The value of this expression is the system name for the resource desired (par. 9.2.2.1).

Example:

```
var answer: string [1Ø];
    inputfile: text;
begin writeln(output, 'what input file?');
     readln(input, answer);
     reset(inputfile, answer)
end;
```

This technique is also useful when a program wishes to refer always to the same resource without requiring it to be specified interactively each time the program is run.

Example:
```
        var commoutput: file of commrecord;
        begin rewrite(commoutput, 'file27')
        end;
```

9.2.2.1 Resource Name String. The resource name string for the M6809 consists of an MDOS device name or file name, optionally followed by a semicolon (;) and sequence of letters and/or digits forming the option specification.

9.2.2.1.1 Syntax. (Refer to Appendix A.)

```
<resource name string> ::= <name>[;[<option>]...]

<name> ::= <device name>[:<lu>]|<filename>[.<suffix>][:<lu>]

<device name> ::= #<device mnemonic>

<device mnemonic> ::= CN|LP|CP|CR

<lu> ::= <digit>

<filename> ::= <identifier>

<suffix> ::= <identifier>

<option> ::= W|D|S|C|N|Ø|1|2|3|5|7|F|R|I<digit sequence>|,
```

9.2.2.1.2 Option Specification. The following file-attribute options are defined:

W - Write protection

D - Delete protection

S - System attribute

C - Contiguous allocation

N - Non-compression ASCII spaces

The following file formats are defined:

Ø- User-defined records. Sector I/O required.

1 - Binary records. Defaults to format 3 or 7, depending upon the device.

2 - Memory-imaged. Sector I/O required.

3 - Binary records. (8-bit data bytes)

5 - ASCII records.

7 - ASCII-converted binary records. (7-bit data bytes).

Other options defined:

F - Forces file-mode I/O for non-diskette devices. (Default for non-diskette devices is non-file mode.)

R - Forces record I/O, overriding the default sector I/O, when the component size is a multiple of the sector size (128 bytes).
I <number> - initial sector allocation for a new file.

The following default values are utilized:

Device type	= DK
Logical unit	= Ø
Filename	= PFxxxx.SY
File format	= Ø (if non-text and sector I/O)
	= 3 (if non-text and record I/O)
	= 5 (if text)
File attribute flags	= Ø (off) for all flags
File/Non-file flag	= file mode
Record/sector flag	= record (if component size is not multiple of sector size)
	= sector (if component size is multiple of sector size)
Sector allocation	= 128 sectors

Examples of resource name strings:

```
FILE1.SA
SAM.SA:1; DI24
CRT.CM:1; SCI48,2
#LP; 5
#CR; F7
```

9.3 OPERATIONS COMMON TO ALL FILE TYPES

In the following descriptions, we use the following conventions:

f is a file identifier.
v, v1, ..., vn are variables of the file's component type.
e, e1, ..., en are expressions of the file's component type.
se is an expression of string type.
i, j, k are expressions of type integer.

9.3.1 Functions

position(f)	Returns an integer denoting the current position of the "read/write head" in file f. The position of the first component in f is 1.
eof(f)	Returns the Boolean value true if the position of file f is past its last component; false otherwise.

9.3.2 Procedures

put(f)	Eof(f) must be true or a runtime error occurs and the program aborts. The value of buffer variable f@ is appended to the end of the file. Position(f) is advanced by 1. Eof(f) remains true.
get(f)	If eof(f) is true, a runtime error occurs and the program aborts. Position(f) is advanced by 1. If this is beyond the sequence of file components, eof(f) is set true and f@ becomes undefined. Otherwise, eof(f) remains false and f@ receives the value of the component at the new file position.
put	Is equivalent to put(output)
get	Is equivalent to get(input)
write(f, e)	Is equivalent to f@ := e; put(f)
write (f, e1, ..., en)	Is equivalent to write(f, e1); ...; write(f, en)
write(e)	Is equivalent to write(output, e)
read(f, v)	Is equivalent to v := f@ ; get(f)
read (f, v1, ... vn)	Is equivalent to read(f, v1); ...; read(f, vn)
read(v)	Is equivalent to read(input, v)
reset(f)	Position(f) becomes 1. If f is empty, eof(f) becomes true and f@ becomes undefined. Otherwise, eof(f) becomes false and f@ refers to the first component of f. Reset (input) is not required since it is automatically generated.
rewrite(f)	Position(f) becomes 1. Length(f) becomes Ø. Eof(f) becomes true. Any previous components of f are discarded; f may now be constructed with entirely new data. Rewrite (output) is not required since it is automatically generated.
reset(f, se), rewrite(f, se)	Behave like reset(f) and rewrite(f), respectively, except that f loses any previous assignment to an I/O resource and becomes assigned to the resource whose name is the value of the string expression se.

If an activation of the procedure put(f) is not separated dynamically from a previous activation of get(f) or reset(f) by an activation of rewrite(f), a runtime error will occur.

9.4 OPERATIONS ON TEXTFILES

The basis of legible input and output consists of textfiles that represent some input or output device such as a terminal or printer. In order to facilitate input and output to textfiles, the standard procedures get and put are augmented with special versions of read, write, readln, and writeln, which have inherent formatting capabilities. The parameters of these procedures need not necessarily be of type char, but may also be of certain other types, in which case implicit conversions are done. In addition, the character field width of each item in the file may be controlled on output.

There are two predeclared files of type text. They are called input and output; if the file identifier is omitted in any get, put, read, write, readln, writeln, eof, or eoln call, the appropriate predeclared file will be assumed.

Textfiles represent a special case among file types insofar as texts are substructured into lines by so-called line markers. If, upon reading a textfile f, the file position is advanced to a line marker -- i.e., past the last character of a line -- then the value of the buffer variable f@ becomes a blank, and the standard function eoln(f) (end of line) yields the value true. Advancing the file position once more assigns to f@ the first character of the next line, and eoln (f) yields false (unless the next line consists of 0 characters). Line markers, not being elements of type char, can only be generated by the procedure writeln and sensed by the function eoln and the procedure readln.

9.4.1 Procedures

9.4.1.1 Write. Let pl, ..., pn denote so-called "write parameters".

1. Write(pl, ..., pn) is equivalent to write(output, pl, ..., pn).

2. The write parameters pi have the following forms:

   e e:i

   e represents the value to be "written" on the file f, and i is a so-called field width parameter. If the value e, which is either a number, a character, a scalar value, or a string, requires less than i characters for its representation, then an adequate number of blanks is issued such that exactly i characters are written. If i is omitted, an implementation-defined default width will be assumed.

3. If e is of type char, then

   ```
   write(f, e:i) is equivalent to
   f@ := ' ' ; put(f); (repeated i-1 times)
   f@ := e; put(f)
   ```

   NOTE: the default value for i is in this case 1.

4. If e is of type integer (or a subrange of integer), then the decimal representation of the number e will be written on the file f, preceded by an appropriate number of blanks to make the field width i.

5. If e is of type Boolean, then a representation of the word true or the word false, as appropriate, is written on the file f. This is equivalent to

   ```
   write(f, 'TRUE ': i)  or  write(f,'FALSE': i)
   ```

   as appropriate (see rule 6).

6. If e is an array of char or of _string_ type, then e is written on the file f, preceded by an appropriate number of blanks to make the field width i.

7. If k stands for the minimum number of characters to properly represent an output value, then the number of characters actually emitted for that item is the greater of k and i.

9.4.1.2 _Writeln_.

1. writeln(p1, ..., pn) is equivalent to writeln(output, p1, ..., pn).

2. writeln(f, p1, ..., pn) is equivalent to write(f, p1, ..., pn); writeln(f).

3. writeln(f) appends a line marker to the file f.

9.4.1.3 _Read_.

1. Let v, v1, ..., vn be simple types (scalar, standard, or subrange). Then:

   read(v1, ..., vn) is equivalent to read(input, v1, ..., vn).

2. If v is of type char, exactly one character is obtained from the file and assigned to v.

3. If v is of a simple type but not char, a sequence of characters is obtained from f which form a constant of the type of v, which is then assigned to v. The first character that does not conform to the syntax of the type of v terminates the input process, including the end-of-line marker. If what was read is not a valid constant of the type of v, a runtime I/O error occurs.

9.4.1.4 _Readln_.

1. readln(v1, ..., vn) is equivalent to readln(input, v1, ..., vn).

2. readln(f, v1, ..., vn) is equivalent to read(f, v1, ..., vn);
   readln(f).

3. readln(f) is equivalent to

   ```
   while not eoln(f) do get(f);
   get(f)
   ```

   Readln is used to read and subsequently skip to the beginning of the next line.

9.4.1.5 Page.

Page(f) causes skipping to the top of a new page, when the textfile f is printed.

9.4.2 Functions

eoln(f)	Returns the Boolean value true if and only if textfile f is positioned at an end-of-line marker. When eoln(f) = true, f@ contains a blank.

CHAPTER 10

SEPARATE COMPILATION AND LINKAGE

10.1 PASCAL SUBPROGRAMS <subprogram>

Procedure and function declarations may be labeled forward in the Pascal program, and their specific program parts may be treated as subprograms and compiled separately. The subprograms are linked to the main program using the M6809 Linking Loader (RLOAD). Figure 10-1 represents the Pascal linkage process.

In order to preserve recognition of global names, all variables in a subprogram must agree in type, number, and order with those appearing in the program's declaration part. The program's variables are global to all compilations.

Labels, constants, and types in a program may be duplicated in a subprogram.

10.2 ASSEMBLY LANGUAGE ROUTINES

Assembly language routines may be linked to the Pascal main program using the M6809 Linking Loader (RLOAD).

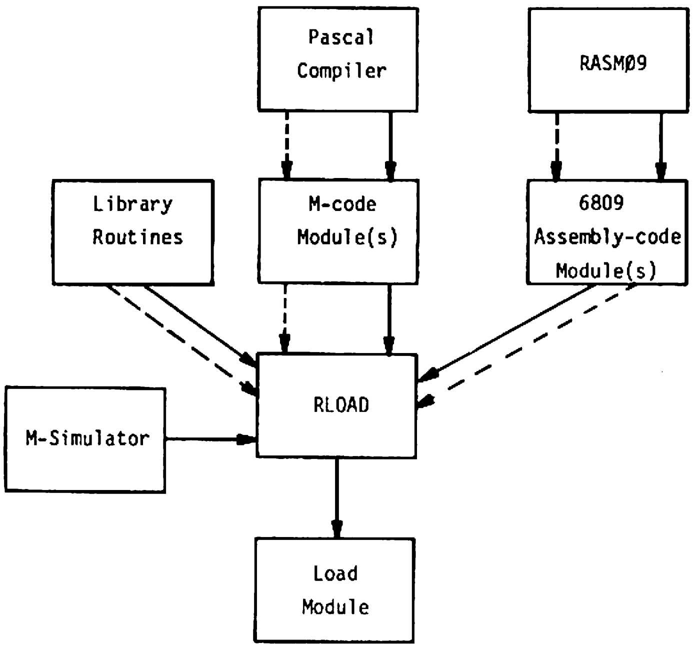

FIGURE 10-1. Pascal Linkage Process

APPENDIX A

SYNTAX

Throughout the manual, a syntactic apex variable is given for each Pascal segment. (For example, <assignment statement> is the apex variable for the assignment statement (7.1.1).) Starting from the apex, the complete format of the syntax may be defined by analyzing the syntactic variable(s) and constant(s) on the right of the define symbol (::=). Right-hand variables refer to other left-hand variable entries in the alphabetical listing.

To use this syntax by itself, <source module> is the apex variable to start with. All the syntax is included in the expanded definition of <source module>.

Note -- The following symbols are meta-symbols belonging to the syntactic format, and not symbols of the Pascal programming language itself, except as noted:

< >	Enclose a symbol, called a syntactic variable.
::=	"is defined as"
|	"or"
[ ]...	Denote that the enclosed symbols are optional/repetitive -- that is, occur zero or more times.
[ ]	Denote that the enclosed symbols are optional -- that is, occur zero or one time. Exceptions are in <array type>, <identifier item>, <indexed variable>, <set>, and <string type> definitions, as noted.

```
<actual parameter> ::= <expression>|<variable>|<procedure identifier>|
     <function identifier>

<adding operator> ::= +|-|or

<array type> ::= array [<index type>[,<index type>]...] of <component type>
     NOTE:  The inside brackets are meta-symbols; the outside brackets are
            symbols of the language.

<array variable> ::= <variable>

<assignment statement> ::= <variable> := <expression>|
     <function identifier> := <expression>

<base type> ::= <simple type>

<block> ::= [<declaration part>]...<statement part>

<block or directive> ::= <block>| forward

<case label> ::= <constant>

<case label list> ::= <case label>[,<case label>]...
```

```
<case list element> ::= <case label list>:<statement>|<empty>

<case statement> ::= case <expression> of <case list element>
     [;<case list element>]...[otherwise <statement>[;<statement>]...] end

<character> ::= any displayable ASCII character

<component type> ::= <type>

<component variable> ::= <indexed variable>|<field designator>|<file buffer>

<compound statement> ::= begin <statement>[;<statement>]... end

<conditional statement> ::= <if statement>|<case statement>

<constant> ::= <unsigned number>|<sign><unsigned number>|<constant identifier>|
     <sign><constant identifier>|<string>

<constant definition> ::= <identifier>=<constant>

<constant definition part> ::= const <constant definition>
     [;<constant definition>]...;

<constant identifier> ::= <identifier>

<control variable> ::= <identifier>

<declaration part> ::= <label declaration part>|<constant definition part>|
     <type definition part>|<variable declaration part>|
     <procedure declaration>|<function declaration>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<digit sequence> ::= <digit>[<digit>]...

<element> ::= <expression>|<expression>..<expression>

<element list> ::= <element>[,<element>]...|<empty>

<empty> ::=

<empty statement> ::= <empty>

<entire variable> ::= <variable identifier>

<exit statement> ::= exit [<label>]

<factor> ::= <variable>|<unsigned constant>|<function designator>|<set>|
     (<expression>)| not <factor>

<field designator> ::= <record variable>.<field identifier>

<field identifier> ::= <identifier>

<field list> ::= <fixed part>|<fixed part>;<variant part>|<variant part>
```

```
<file buffer> ::= <file variable>@

<file type> ::= file of <type>

<file variable> ::= <variable>

<final value> ::= <expression>

<fixed part> ::= <record section>[;<record section>]...

<for list> ::= <initial value> to <final value>|<initial value> downto
     <final value>

<formal parameter section> ::= <parameter group>| var <parameter group>|
     function <parameter group> | procedure <identifier>[,<identifier>]...

<formal parameter spec> ::= (<formal parameter section>
     [;<formal parameter section>]...)

<for statement> ::= for <control variable> := <for list> do <statement>

<function declaration> ::= <function heading><block or directive>;

<function designator> ::= <function identifier>|<function identifier>
     (<actual parameter>[,<actual parameter>]...)

<function heading> ::= function <identifier>[<formal parameter spec>]:
     <result type>;

<function identifier> ::= <identifier>

<goto statement> ::= goto <label>

<hexdigit> ::= <digit>|A|B|C|D|E|F

<hexdigit sequence> ::= <hexdigit>[<hexdigit>]...

<identifier> ::= <letter>[<letter or digit>]...

<identifier item> ::= <identifier>[[origin <unsigned integer>]]
     NOTE:  The inside brackets are symbols of the language; the outside
            brackets are meta-symbols.

<if statement> ::= if <expression> then <statement>| if <expression> then
     <statement> else <statement>

<indexed variable> ::= <array variable>[<expression>[,<expression>]...]
     NOTE:  The inside brackets are meta-symbols; the outside brackets are
            symbols of the language.

<index type> ::= <simple type>

<initial value> ::= <expression>

<label> ::= <unsigned integer>|<identifier>
```

```
<label declaration part> ::= label <label>[,<label>]...;

<letter> ::=
     A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|
     l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|_

<letter or digit> ::= <letter>|<digit>

<max length> ::= <unsigned integer>

<multiplying operator> ::= *|div|mod|and

<parameter group> ::= <identifier>[,<identifier>]...:<type identifier>

<pointer type> ::= @<type identifier>

<pointer variable> ::= <variable>

<procedure declaration> ::= <procedure heading><block or directive>;

<procedure heading> ::= procedure <identifier>[<formal parameter spec>];

<procedure identifier> ::= <identifier>

<procedure statement> ::= <procedure identifier>|<procedure identifier>
     (<actual parameter>[,<actual parameter>]...)

<program> ::= <program heading><block>.

<program heading> ::= program <identifier>(<program parameters>);

<program parameters> ::= <identifier>[,<identifier>]...

<record section> ::= <field identifier>[,<field identifier>]...:<type>|<empty>

<record type> ::= record <field list> end

<record variable> ::= <variable>

<record variable list> ::= <record variable>[,<record variable>]...

<referenced variable> ::= <pointer variable>@

<relational operator> ::= =|<>|<|<=|>=|>|in

<repeat statement> ::= repeat <statement>[;<statement>]... until <expression>

<repetitive statement> ::= <while statement>|<repeat statement>|
     <for statement>

<result type> ::= <type identifier>

<scalar type> ::= (<identifier>[,<identifier>]...)

<set> ::= [<element list>]
     NOTE:  The brackets are symbols of the language.
```

```
<set type> ::= set of <base type>

<sign> ::= +|-

<simple expression> ::= <term>|<simple expression><adding operator><term>|
     <sign><term>

<simple statement> ::= <assignment statement>|<procedure statement>|
     <goto statement>|<empty statement>|<exit statement>

<simple type> ::= <scalar type>|<subrange type>|<type identifier>

<source module> ::= <program>|<subprogram>

<statement> ::= <unlabeled statement>|<label>:<unlabeled statement>

<statement part> ::= <compound statement>

<string> ::= '<character>[<character>]...'

<string type> ::= string [<max length>]
     NOTE:  The brackets are symbols of the language.

<structured statement> ::= <compound statement>|<with statement>|
     <conditional statement>|<repetitive statement>

<structured type> ::= <array type>|<record type>|<set type>|
     <file type>|<string type>

<subprogram> ::= <subprogram heading>[<declaration part>]...

<subprogram heading> ::= subprogram <identifier>(<subprogram parameters>);

<subprogram parameters> ::= <identifier>[,<identifier>]...

<subrange type> ::= <constant>..<constant>

<tag field> ::= <identifier>:|<empty>

<term> ::= <factor>|<term><multiplying operator><factor>

<type> ::= <simple type>|<structured type>|<pointer type>

<type definition> ::= <identifier>=<type>

<type definition part> ::= type <type definition>[;<type definition>]...;

<type identifier> ::= <identifier>

<unlabeled statement> ::= <simple statement>|<structured statement>

<unsigned constant> ::= <unsigned number>|<string>|<constant identifier>| nil

<unsigned integer> ::= [<digit sequence>#]<hexdigit sequence>

<unsigned number> ::= <unsigned integer>
```

```
<variable> ::= <entire variable>|<component variable>|<referenced variable>

<variable declaration> ::= <identifier item>[,<identifier item>]...:<type>

<variable declaration part> ::= var <variable declaration>
     [;<variable declaration>]...;

<variable identifier> ::= <identifier>

<variant> ::= <case label list>:(<field list>)|<empty>

<variant part> ::= case <tag field><type identifier> of <variant>[;<variant>]...

<while statement> ::= while <expression> do <statement>

<with statement> ::= with <record variable list> do <statement>
```

****

The following syntax is defined for Pascal features to be included in future releases:

```
<constant definition> ::= <identifier>=<constant>|<identifier>=
     <type identifier>(<constant list>)

<constant list> ::= <constant term>[,<constant term>]...

<constant term> ::= <constant>|(<constant list>)|<unsigned integer>
     of (<constant list>)

<file type> ::= file of <type>| indexed file of <type>

<formal parameter section> ::= <parameter group>| var <parameter group>|
     function <identifier>[,<identifier>]...[<formal parameter spec>]:
     <type identifier>| procedure <identifier>[,<identifier>]...
     [<formal parameter spec>]

<multiplying operator> ::= *|/|div|mod|and

<scale factor> ::= [<sign>]<digit sequence>

<structured type> ::= <unpacked structured type>|
     packed <unpacked structured type>

<unpacked structured type> ::= <array type>|<record type>|<set type>|
     <file type>|<string type>

<unsigned number> ::= <unsigned integer>|<unsigned real>

<unsigned real> ::= <digit sequence>.<digit sequence> E <scale factor>|
     <digit sequence>.<digit sequence>|<digit sequence> E <scale factor>
```