MICROSOFT®

The High Performance Software™

# Microsoft® Pascal

## Reference Manual

Microsoft Corporation

If you have comments about the software or these manuals, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Document Number 8501-330-08
Part Number 020-014-027

# Contents

# Figures

# Tables

# How to Use This Manual

Chapter 1, "Introduction to Microsoft Pascal," introduces the reader to the selected and unimplemented features of the Microsoft Pascal language and gives a brief overview of the standard, extend, and system levels of Microsoft Pascal programming.

Chapter 2, "Language Overview," paints a broad picture of Microsoft Pascal, from the largest elements of the language to the smallest.

Chapters 3 through 11 describe the elements of the language including notation and identifier conventions, conventions for data transfer (file I/O), available data structures, and the data types through which these data structures are accessed.

Chapters 12 and 13 define the expressions and statements that are supported by Microsoft Pascal.

Chapters 14 through 18 describe the procedures, functions, and metalanguage supported by Microsoft Pascal.

Appendices A through G contain supplementary material including a discussion of Microsoft Pascal features, and summaries of Microsoft Pascal procedures, functions, metalanguage, and syntax.

# Chapter 1
# Introduction to Microsoft Pascal

Microsoft® Pascal offers systems programmers a clean, structured language that is especially adept at quickly handling complex financial and scientific algorithms. By generating native code, Microsoft Pascal offers the advantages of a high level programming language without sacrificing speed. Low level escapes to the machine level allow Microsoft Pascal programs to achieve speeds comparable to assembly language.

Microsoft Pascal offers fast numeric processing in the 8087 processing environment and also provides 8087 emulation in the system software package.

## 1.1 Levels of Microsoft Pascal

MS™-Pascal is organized into three levels: standard, extend, and system. The features of each level are discussed in more detail in Appendix B, "Microsoft Pascal Features and the ISO Standard." Briefly, the differences among the three levels are as follows:

1. Standard level

   At the standard level, programs must conform to the ISO standard. Programs you create at this level are portable to and from machines running other ISO-compatible Pascal compilers. There are some minor MS-Pascal extensions to the standard that won't be caught as errors at this level. For details of these extensions, as well as other issues regarding the ISO standard, see Appendix B, "Microsoft Pascal Features and the ISO Standard." In this manual, the phrases "standard Pascal," "the ISO standard," and "at the standard level of MS-Pascal" are generally synonymous.

2. Extend level

   The extend level is intended for structured and relatively safe extensions to the ISO standard. Programs you create at this level are portable among all machines that run MS-Pascal.

3. System level

   The system level includes all features available at the extend level. It also includes some unstructured, machine-oriented extensions, such as address types and the ability to access all file control block fields, which are useful for systems programming.

In this manual, extensions to standard Pascal are called "features." A complete list of these features and the level at which they are available is given in Appendix B, "Microsoft Pascal Features and the ISO Standard." Selected features are described briefly in the following list.

In addition to these three levels, MS-Pascal has a large number of "metacommands," that is, directives with which you can control the compiler. See Chapter 18, "Microsoft Pascal Metacommands," for more information.

## 1.2 Selected Features

The following list includes several of the features available at the extend and system levels of MS-Pascal. For a complete list, see Section B.2, "Summary of Microsoft Pascal Features."

1. Underscore in identifiers, which improves readability.
2. Nondecimal numbering (hexadecimal, octal, and binary), which facilitates programming at the byte and bit level.
3. Structured constants, which you may declare in the declaration section of a program or use in statements.
4. Variable length strings (type LSTRING), as well as special predeclared procedures and functions for LSTRINGs, which enhance standard Pascal's string handling capabilities.
5. Super arrays, which are special variable length arrays whose declaration permits passing arrays of different lengths to a reference parameter, as well as dynamic allocation of arrays of different lengths.

6. Predeclared unsigned BYTE (0-255) and WORD (0-65535) types, which facilitate programming at the system level.
7. Address types (segmented and unsegmented), which allow manipulation of actual machine addresses at the system level.
8. String reads, which allow the standard procedures READ and READLN to read strings as structures rather than character by character.
9. Interface to assembly language, provided by PUBLIC and EXTERN procedures, functions, and variables, which allows low-level interfacing to assembly language and library routines.
10. VALUE section, where you may declare the initial constant values of variables in a program.
11. Function return values of a structured type as well as of a simple type.
12. Direct (random access) files, accessible with the SEEK procedure, which enhance standard Pascal's file accessing capabilities.
13. Lazy evaluation, a special internal mechanism for interactive files which allows normal interactive input from terminals.
14. Structured BREAK and CYCLE statements, which allow structured exits from a FOR, REPEAT, or WHILE loop; and the RETURN statement, which allows a structured exit from a procedure or function.
15. OTHERWISE in CASE statements, whereby you avoid explicitly specifying each CASE constant. OTHERWISE is also permitted with variant records.
16. STATIC attribute for variables, which allows you to indicate that a variable is to be allocated at a fixed location in memory rather than on the stack.
17. ORIGIN attribute, which may be given to variables, procedures, and functions to indicate their absolute location in memory.

18. INTERRUPT attribute for procedures, which signals the compiler to give the procedure a special calling sequence that saves the machine status on the stack upon entry and restores the machine status upon exit.
19. Separate compilation of portions of a program (units and modules).
20. Conditional compilation, which uses conditional metacommands in your MS-Pascal source file to switch on or off compilation of parts of the source.

## 1.3 Unimplemented Features

The following features either are not presently implemented or are implemented only as described in the following list:

1. OTHERWISE is not accepted in RECORD declarations.
2. Code is generated for PURE functions, but no checking is done.
3. The extend level operators SHL, SHR, and ISR are not available.
4. No checking is done for invalid GOTOs and uninitialized REAL values.
5. READ, READLN, and DECODE cannot have M and N parameters.
6. Enumerated I/O, for reading and writing enumerated constants as strings, is not available.
7. The metacommands $tagck, $standard, $extend, and $system can be given, but have no effect.
8. The $inconst metacommand does not accept string constants.

# Chapter 2
# Language Overview

In this chapter you will find a summary description of Microsoft Pascal from the largest elements of the language to the smallest. Each of the remaining chapters of the manual discusses these elements in detail, from the smallest element (notation) to the largest (metacommands).

## 2.1 Metacommands

The MS-Pascal metacommands provide a control language for the MS-Pascal Compiler. The metacommands let you specify options that affect the overall operation of a compilation. For example, you can conditionally compile different source files, generate a listing file, or enable or disable runtime error checking code.

Metacommands are inserted inside comment statements. All of the metacommands begin with a dollar sign ($). Some may also be given as switches when you invoke the compiler.

Although most implementations of Pascal have some type of compiler control, the MS-Pascal metacommands are not part of standard Pascal and hence are not portable.

These are the metacommands currently available:

```
$brave                  $linesize     $rom
$debug                  $list         $runtime
$decmath                $mathck       $simple
$entry                  $message      $size
$errors                 $nilck        $skip
$extend                 $ocode        $speed
$floatcalls             $optbug       $stackck
$goto                   $page         $standard
$if $then $else $end    $pageif       $subtitle
$include                $pagesize     $symtab
$inconst                $pop          $system
$indexck                $push         $tagck
$initck                 $rangeck      $title
$line                   $real         $warn
```

See Chapter 18, "Microsoft Pascal Metacommands," for a complete discussion of metacommands.

## 2.2 Programs and Compilable Parts of Programs

The MS-Pascal Compiler processes programs, modules, and implementations of units. Collectively, these compilable programs and parts of programs are referred to as compilands. You can compile modules and implementations of units separately and then later link them to a program without having to recompile the module or unit.

The fundamental unit of compilation is a program. A program has three parts:

1. The program heading identifies the program and gives a list of program parameters.
2. The declaration section follows the program heading and contains declarations of labels, constants, types, variables, functions, and procedures. Some of these declarations are optional.
3. The body follows all declarations. It is enclosed by the reserved words BEGIN and END and is terminated by a period. The period is the signal to the compiler that it has reached the end of the source file.

The following program illustrates this three-part structure:

```
{Program heading}
PROGRAM FRIDAY (INPUT, OUTPUT);

{Declaration section}
LABEL 1;
CONST DAYS_IN_WEEK = 7;
TYPE KEYBOARD_INPUT = CHAR;
VAR KEYIN : KEYBOARD_INPUT;
```

```
{Program body}
BEGIN
     WRITE('IS TODAY FRIDAY? ');
1:   READLN(KEYIN);
     CASE KEYIN OF
      'Y', 'y' : WRITELN('It''s Friday.');
      'N', 'n' : WRITELN('It''s not Friday.');
     OTHERWISE
          WRITELN('Enter Y or N.');
          WRITE('Please re-enter: ');
          GOTO 1
     END
END.
```

This three-part structure (heading, declaration section, and body) is used throughout the Pascal language. Procedures, functions, modules, and units are all similar in structure to a program.

Modules are program-like units of compilation that contain the declaration of variables, constants, types, procedures, and functions, but no program statements. You can compile a module separately and later link it to a program, but it cannot be executed by itself.

Example of a module:

```
{Module heading}
MODULE MODPART;

{Declaration section}
CONST PI = 3.14;

PROCEDURE PARTA;
     BEGIN
        WRITELN ('parta')
     END;

{Body}
END.
```

A module, like a program, ends with a period. Unlike a program, a module contains no program statements.

A unit has two sections: an interface and an implementation. Like a module, an implementation may be compiled separately and later linked to the rest of the program. The interface contains the information that lets you connect a unit to other units, modules, and programs.

Example of a unit:

```
{Heading for interface}
INTERFACE;
UNIT MUSIC (SING, TOP);

{Declarations for interface}
VAR TOP : INTEGER;
PROCEDURE SING;

{Body of interface}
BEGIN
END;

{Heading for implementation}
IMPLEMENTATION OF MUSIC;

{Declaration for implementation}
PROCEDURE SING;
BEGIN
      FOR I := 1 TO TOP DO
      BEGIN
          WRITE ('FA'); WRITELN ('LA LA')
      END
END;

{Body of implementation}
BEGIN
      TOP := 5
END.
```

A unit, like a program or a module, ends with a period.

Modules and units let you develop large structured programs that can be broken into parts. This practice is advantageous in the following situations:

1. If a program is large, breaking it into parts makes it easier to develop, test, and maintain.
2. If a program is large and recompiling the entire source file is time consuming, breaking the program into parts saves compilation time.

3. If you intend to include certain routines in a number of different programs, you can create a single object file that contains these routines and then link it to each of the programs in which the routines are used.
4. If certain routines have different implementations, you might place them in a module to test the validity of an algorithm and later create and implement similar routines in assembly language to increase the speed of the algorithm.

See Chapter 17, "Compilable Parts of a Program," for a complete discussion of programs, modules, and units.

## 2.3 Procedures and Functions

Procedures and functions act as subprograms that execute under the supervision of a main program. However, unlike programs, procedures and functions can be nested within each other and can even call themselves. Furthermore, they have sophisticated parameter-passing capabilities that programs lack.

Procedures are invoked as statements; functions can be invoked in expressions wherever values are called for.

A procedure declaration, like a program, has a heading, a declaration section, and a body.

Example of a procedure declaration:

```
{Heading}
PROCEDURE COUNT_TO (NUM : INTEGER);

{Declaration section}
VAR I : INTEGER;

{Body}
BEGIN
   FOR I := 1 TO NUM DO WRITELN (I)
END;
```

A function is a procedure that returns a value of a particular type; hence, a function declaration must indicate the type of the return value.

Example of a function declaration:

```
{Heading}
FUNCTION ADD (VAL1, VAL2 : INTEGER) : INTEGER;

{Declaration section empty}

{Body}
BEGIN
     ADD := VAL1 + VAL2
END;
```

Procedures and functions look somewhat different from programs, in that their parameters have types and other options. Like the body of a program, the body of a procedure or a function is enclosed by the reserved words BEGIN and END; however, a semicolon rather than a period follows the word "END".

Declaring a procedure or function is entirely distinct from using it in a program.

For example, the procedure and function declared above might actually appear in a program as follows:

```
TARGET_NUMBER := ADD (5, 6);
COUNT_TO (TARGET_NUMBER);
```

See Chapter 14, "Introduction to Procedures and Functions," for a complete discussion of procedures and functions.

See Chapter 15, "Available Procedures and Functions," and Chapter 16, "File-Oriented Procedures and Functions," for a discussion of procedures and functions that are predeclared as part of the MS-Pascal language.

## 2.4 Statements

Statements perform actions, such as computing, assigning, altering the flow of control, and reading and writing files. Statements are found in the bodies of programs, procedures, and functions and are executed as a program runs. MS-Pascal statements perform the actions shown in Table 2.1.

**Table 2.1**

**Summary of Microsoft Pascal Statements**

| Statement | Purpose |
|---|---|
| Assignment | Replaces the current value of a variable with a new value |
| BREAK | Exits the currently executing loop |
| CASE | Allows for the selection of one action from a choice of many, based on the value of an expression |
| CYCLE | Starts the next iteration of a loop |
| FOR | Executes a statement repeatedly while a progression of values is assigned to a control variable |
| GOTO | Continues processing at another part of the program |
| IF | Together with THEN and ELSE, allows for conditional execution of a statement |
| Procedure call | Invokes a procedure with actual parameter call values |
| REPEAT | Repeats a sequence of statements one or more times, until a Boolean expression becomes true |
| RETURN | Exits the current procedure, function, program, or implementation |
| WHILE | Repeats a statement zero or more times, until a Boolean expression becomes false |
| WITH | Opens the scope of a statement to include the fields of one or more records, so that you can refer to the fields directly |

See Chapter 13, "Statements," for a detailed discussion of each of these statements.

## 2.5 Expressions

An expression is a formula for computing a value. It consists of a sequence of operators (which indicate the action to be performed) and operands (which are the value on which the operation is performed). Operands may contain function invocations, variables, constants, or even other expressions. In the following expression, plus (+) is an operator, while A and B are operands:

```
A + B
```

There are three basic kinds of expressions:

1. Arithmetic expressions perform arithmetic operations on the operands in the expression.
2. Boolean expressions perform logical and comparison operations with Boolean results.
3. Set expressions perform combining and comparison operations on sets, with Boolean or set results.

Expressions always return values of a specific type. For instance, if A, B, C, and D are all REAL variables, then the following expression evaluates to a REAL result:

```
A * B + (C / D) + 12.3
```

Expressions may also include function designators:

```
ADDREAL (2, 3) + (C / D)
```

ADDREAL is a function that has been previously declared in a program. It has two REAL value parameters, which it adds together to obtain a total. This total is the return value of the function, which is then added to (C / D).

Expressions are not statements, but may be components of statements. In the following example, the entire line is a statement; only the portion after the equal sign is an expression:

```
X := 2 / 3 + A * B
```

See Chapter 12, "Expressions," for a detailed discussion of expressions.

## 2.6 Variables

A variable is a value that is expected to change during the course of a program. Every variable must be of a specific data type.

After you declare a variable in the heading or declaration section of a compiland, procedure, or function, it may be used in any of the following ways:

1. You may initialize it, in the VALUE section of a program.
2. You may assign it a value, with an assignment statement.
3. You may pass it as a parameter to a procedure or function.
4. You may use it in an expression.

The VALUE section is an MS-Pascal feature that applies only to statically allocated variables (variables with a fixed address in memory). First you declare the variables, as shown in the following example:

```
VAR I, J, K, L : INTEGER;
```

Then you assign them initial values in the VALUE section:

```
VALUE I := 1; J := 2; K := 3; L := 4;
```

Later, in statements, the variables can be assigned to, and used as operands in expressions:

```
I := J + K + L;
J := 1 + 2 + 3;
K := (J * K) + 9 + (L DIV J);
```

See Chapter 11, "Variables and Values," for a complete discussion of variables.

## 2.7 Constants

A constant is a value that is not expected to change during the course of a program. At the standard level, a constant may be:

1. a number, such as 1.234 and 100
2. a string enclosed in single quotation marks, such as 'Miracle' or 'A1207'
3. a constant identifier that is a synonym for a numeric or string constant

You declare constant identifiers in the CONST section of a compiland, procedure, or function.

```
CONST REAL_CONST = 1.234;
      MAX_VAL = 100;
      TITLE = 'Pascal';
```

Because the order of declarations is flexible in MS-Pascal, you can declare constants anywhere in the declaration section of a compilable part of a program, any number of times.

Constants are closely tied to the concepts of variables and types. Variables are all of some type; types, in turn, designate a range of assumable values. These values, ultimately, are all constants.

Two powerful extensions in MS-Pascal are structured constants and constant expressions.

1. VECTOR, in the following example, is an array constant:

   ```
   CONST VECTOR = VECTORTYPE (1, 2, 3, 4, 5);
   ```

2. MAXVAL, in the following example, is a constant expression (A, B, C, and D must also be constants):

   ```
   CONST MAXVAL = A * (B DIV C) + D - 5;
   ```

See Chapter 10, "Constants," for a complete discussion of these and other aspects of constants.

## 2.8 Types

Much of MS-Pascal's power and flexibility lies in its data typing capability. Although a great variety of data types are available, they may be divided into three broad categories: simple, structured, and reference types.

1. A simple data type represents a single value. The simple types include the following:

   | | |
   |---|---|
   | INTEGER | enumerated |
   | WORD | subrange |
   | CHAR | REAL |
   | BOOLEAN | INTEGER4 |

2. A structured data type represents a collection of values. The structured types include the following:

   ARRAY

   RECORD

   SET

   FILE

3. Reference types allow recursive definition of types.

All variables in MS-Pascal must be assigned a data type. A type is either predeclared (e.g., INTEGER and REAL) or defined in the declaration section of a program. The following sample type declaration creates a type that can store information about a student:

```
TYPE
    STUDENT = RECORD
        AGE : 5 . . 18;
        SEX : (MALE, FEMALE);
        GRADE : INTEGER;
        GRADE_PT : REAL;
        SCHEDULE : ARRAY [1 . . 10] OF CLASSES
    END;
```

For a detailed discussion of data types, see the following chapters:

Chapter 5, "Introduction to Data Types"

Chapter 6, "Simple Types"

Chapter 7, "Arrays, Records, and Sets"

Chapter 8, "Files"

Chapter 9, "Reference and Other Types"

## 2.9 Identifiers

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program. Procedures and functions must have identifiers; constants, type, and variables may have identifiers (and it is useful if they do).

You, the programmer, make up most of the identifiers in a program and assign them meaning in declarations. Other identifiers are the names of variables, data types, procedures, and functions that are built into the language and need not be declared.

An identifier must begin with a letter (A through Z and a through z). The initial letter may be followed by any number of letters, digits (0 through 9), or underscore characters.

The underscore in MS-Pascal is significant. Thus, the following are not identical:

```
FOREST
FOR_EST
```

The only restriction on identifiers is that you must not choose a Pascal reserved word (see Section 3.3.3, "Reserved Words," for a discussion of reserved words; see Appendix E, "Summary of Microsoft Pascal Reserved Words," for a complete list).

Furthermore, most compilers have some restriction either on the absolute length of an identifier or on the number of characters that are considered significant.

See Chapter 4, "Identifiers," for a complete discussion of identifiers in MS-Pascal.

## 2.10 Notation

The basis of all Pascal programs is an irreducible set of symbols with which the higher syntactic components of the language are created.

The underlying notation is the ASCII character set, divided into the following syntactic groups:

1. Identifiers are the names given to individual instances of components of the language.
2. Separators are characters that delimit adjacent numbers, reserved words, and identifiers.
3. Special symbols include punctuation, operators, and reserved words.
4. Some characters are unused by MS-Pascal but are available for use in a comment or string literal.

A good understanding of this notation will increase your productivity by reducing the number of subtle syntactic errors in a program. See Chapter 3, "Notation," for a detailed discussion of MS-Pascal notation.

# Chapter 3
# Notation

All components of the Microsoft Pascal language are constructed from the standard ASCII character set. Characters make up lines, each of which is separated by a character specific to your operating system. Lines make up files.

Within a line, individual characters or groups of characters fall into one (or more) of four broad categories:

1. components of identifiers
2. separators
3. special symbols
4. unused characters

## 3.1 Components of Identifiers

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program.

The use of identifiers is described thoroughly in Chapter 4, "Identifiers." This section discusses only how to construct them.

Identifiers must begin with a letter; subsequent components may include letters, digits, and underscore characters.

Although, in theory, there is no limit on the number of characters in an identifier, most implementations restrict the length in some way.

### 3.1.1 Letters

In identifiers, only the uppercase letters A through Z are significant. You may use lowercase letters for identifiers in a source program. However, the MS-Pascal Compiler converts all lowercase letters in identifiers to the corresponding uppercase letters.

Letters in comments or in string literals may be either uppercase or lowercase; the difference is significant. No mapping of lowercase to uppercase occurs in either comments or string literals.

### 3.1.2 Digits

Digits in Pascal are the numbers zero through nine. Digits can occur in identifiers (for example, AS129M) or in numeric constants (for example, 1.23 and 456).

### 3.1.3 The Underscore Character

The underscore (_) is the only nonalphanumeric character allowed in identifiers. The underscore is significant in MS-Pascal. Use it like a space to improve readability.

The identifiers in the left-hand column below demonstrate how the underscore improves readability. Note, however, that they will not be equal to those in the right-hand column.

```
POWER_OF_TEN        POWEROFTEN
MY_DOG_MAUDE        MYDOGMAUDE
```

## 3.2 Separators

Separators delimit adjacent numbers, reserved words, and identifiers, none of which should have separators embedded within them. A separator can be any of the following:

1. the space character
2. the tab character
3. the form feed character
4. the new line marker
5. the comment

Comments in standard Pascal take one of these forms:

```
{This is a comment, enclosed in braces.}
(*This is an alternate form of comment.*)
```

The second form is generally used if braces are unavailable on a particular machine. Comments in either of these forms may span more than one line.

At the extend level, MS-Pascal also allows comments that begin with an exclamation point:

```
! The rest of this line is a comment.
```

For comments in this form, the new line character delimits the comment.

Nested comments are permitted in MS-Pascal, so long as each level has different delimiters. Thus, when a comment is started, the compiler ignores succeeding text until it finds the matching end-of-comment. However, such nesting may not be portable.

Always use separators between identifiers and numbers. If you fail to do so, the compiler will generally issue an error or warning message. In a few cases, the MS-Pascal Compiler accepts a missing separator without generating an error message.

For example, at extend level,

```
100MOD#127
```

is accepted as 100 MOD #127, where #127 is a hexadecimal number. However,

```
100MOD127
```

is assumed to be 100 followed by the identifier MOD127.

## 3.3 Special Symbols

Special symbols fall into three categories:

1. punctuation

2. operators
3. reserved words

## 3.3.1 Punctuation

Punctuation in MS-Pascal serves a variety of purposes, including those shown in Table 3.1.

**Table 3.1**

**Summary of Punctuation in Microsoft Pascal**

| Symbol | Purpose |
|---|---|
| { } | Braces delimit comments. |
| [ ] | Brackets delimit array indices, sets, and attributes. They may also replace the reserved words BEGIN and END in a program. |
| ( ) | Parentheses delimit expressions, parameter lists, and program parameters. |
| ' | Single quotation marks enclose string literals. |
| := | The colon-equal symbol assigns values to variables in assignment statements and in VALUE sections. |
| ; | The semicolon separates statements and declarations. |
| : | The colon separates variables from types and labels from statements. |
| = | The equal sign separates identifiers and type clauses in a TYPE section. |
| , | The comma separates the components of lists. |
| .. | The double period denotes a subrange. |
| . | The period designates the end of a program, indicates the fractional part of a real number, and also delimits fields in a record. |
| ^ | The up arrow denotes the value pointed to by a reference value. |
| # | The number sign denotes nondecimal numbers. |
| $ | The dollar sign prefixes metacommands. |

## 3.3.2 Operators

Operators are a form of punctuation that indicate some operation to be performed. Some are alphabetic, others are one or two nonalphanumeric characters. Any operators that consist of more than one character must not have a separator between characters.

The operators that consist only of nonalphabetic characters are the following:

```
+  -  *  /  >  <  =  <>  <=  >=
```

Some operators (e.g., NOT and DIV) are reserved words instead of nonalphabetic characters.

See Chapter 12, "Expressions," for a complete list of of the nonalphabetic operators and a discussion of the use of operators in expressions.

## 3.3.3 Reserved Words

Reserved words are a fixed part of the MS-Pascal language. They include, for example, statement names (e.g., BREAK) and words like BEGIN and END that bracket the main body of a program. See Appendix E, "Summary of Microsoft Pascal Reserved Words," for a complete list.

You cannot create an identifier that is the same as any reserved word. You may, however, declare an identifier that contains within it the letters of a reserved word (for example, the identifier DOT containing the reserved word DO).

There are several categories of reserved words in MS-Pascal:

1. reserved words for standard level MS-Pascal
2. reserved words added for extend level MS-Pascal features
3. reserved words added for system level MS-Pascal features
4. names of attributes
5. names of directives

See Appendix E, "Summary of Microsoft Pascal Reserved Words," for a complete list of reserved words. Look in the index for the pages where each is discussed in this manual.

## 3.4 Unused Characters

A few printing characters are not used in MS-Pascal:

```
% & " | ~
```

You may, however, use them within comments or string literals.

A number of other nonprinting ASCII characters will generate error messages if you use them in a source file other than in a comment or string literal:

1. the characters from CHR (0) to CHR (31), except the tab and form feed, CHR (9) and CHR (12), respectively
2. the characters from CHR (127) to CHR (255)

The tab character, CHR (9), is treated like a space and is passed on to the listing file. A form feed, CHR (12), is treated like a space and starts a new page in the listing file.

## 3.5 Notes on Characters

This section discusses special notational properties of the MS-Pascal language not mentioned elsewhere in this chapter. Characters within a comment or string literal are always legal and have no special effects.

MS-Pascal allows the following substitutions:

| If Your Keyboard Lacks | Use This Instead |
|---|---|
| [ | (. |
| ] | .) |
| ^ | @ or ? |
| @ | ^ or ? |

The substitution of a question mark (?) for an up arrow (^) is a minor extension to the ISO standard.

Table 3.2 gives a list of pairs of printing characters that are the same ASCII character.

**Table 3.2**

**Equivalent ASCII Characters**

| ASCII | Prints as | Equivalent Characters |
|---|---|---|
| CHR (94) | ^ | up arrow, caret |
| CHR (95) | _ | underscore, left arrow |
| CHR (35) | # | number sign, English pound sign |
| CHR (36) | $ | dollar sign, scarab (circle with four spikes) |

# Chapter 4
# Identifiers

Procedures and functions must have identifiers. Constants, types, and variables may have identifiers and it is useful if they do. Some identifiers are predeclared; others you declare in a declaration section.

## 4.1 Creating an Identifier

Standard Pascal allows identifiers for the following elements of the Pascal language:

- types
- constants
- variables
- procedures
- functions
- programs
- fields and tag fields in records

The following Microsoft Pascal features at the extend level also allow identifiers:

- super array types
- modules
- units
- statement labels

An identifier consists of a sequence of alphanumeric characters or underscore characters. The first character must be alphabetic. Underscores in identifiers are allowed, and significant, at all levels of MS-Pascal. Two underscores in a row or an underscore at the end of an identifier are permitted.

Subject to the restrictions noted below, identifiers can be as long as you want. They must, however, fit on a single line. At least the first 19 characters of an identifier are significant; in some versions of MS-Pascal, as many as 31 characters are significant.

An identifier longer than the significance length generates a warning but not an error message; the excess characters are ignored by the compiler.

Standard Pascal allows unsigned integers as statement labels. Statement labels have the same scope rules as identifiers (see Section 4.3, "The Scope of Identifiers"). Leading zeros are not significant. Extend level MS-Pascal allows labels that are normal alphabetic identifiers.

The identifiers used for a program, module, or unit, as well as identifiers with the PUBLIC or EXTERN attribute, are passed to the linker. The operating system of a machine on which you plan to link and run a compiled MS-Pascal program may impose further identifier length restrictions on identifiers used as linker global symbols. Furthermore, the object code listing and debugger symbol table may truncate variable and procedural identifiers to six characters.

Writing programs for use with other compilers and operating systems imposes an additional constraint on a program. Such a program must conform to the identifier restrictions for the worst possible case.

For portability in general, the following practices are recommended:

1. Make all identifiers unique in their first eight characters.
2. Make external identifiers unique in their first six characters.
3. Limit statement labels to four digits without leading zeros.

Identifiers of seven characters or fewer save space during compilation.

---

*Note*

All identifiers used internally by the runtime system are four alphabetic characters followed by the characters QQ. Avoid this form when creating new names yourself.

---

## 4.2 Declaring an Identifier

You declare identifiers and associate them with language objects in the declaration section of a program, module, interface, implementation, procedure, or function. Examples of identifiers, the objects they represent, and the syntax used to declare them are shown in Table 4.1. Although the details vary, the basic form of the declaration of the identifier for each of these elements is similar.

**Table 4.1**

**Declaring Identifiers**

| Object | Identifier | Declaration |
|---|---|---|
| Program | Z | PROGRAM Z (INPUT, OUTPUT) |
| Module | XXX | MODULE XXX |
| Interface | UUU | INTERFACE; UNIT UUU |
| Implementation | UUU | IMPLEMENTATION of UUU |
| Constant | DAYS | CONST DAYS = 365 |
| Type | LETTERS | TYPE LETTERS = 'A'..'Z' |
| Record fields | X, Y, Z | TYPE A = RECORD X, Y, Z : REAL END |
| Variable | J | VAR J : INTEGER |
| Label | 1 | LABEL 1 |
| Label | HAWAII | LABEL HAWAII |
| Procedure | BANG | PROCEDURE BANG |
| Function | FOO | FUNCTION FOO : INTEGER |

## 4.3 The Scope of Identifiers

An identifier is defined for the duration of the procedure, function, program, module, implementation, or interface in which you declare it. This holds true for any nested procedures or functions. An identifier's association must be unique within its scope; that is, it must not name more than one thing at a time.

A nested procedure or function can redefine an identifier only if the identifier has not already been used in it. However, the compiler does not identify such redefinition as an error, but will generally use the first definition until the second occurs. A special exception for reference types is discussed in Section 9.1.5, "Notes on Reference Types."

## 4.4 Predeclared Identifiers

A number of identifiers are already a part of the MS-Pascal language. This category includes the identifiers for predeclared types, super array types, constants, file variables, functions, and procedures. You can use them freely, without declaring them. However, they differ from reserved words in that you may redefine them whenever you wish.

At the standard level, the following identifiers are predeclared:

| | | | |
|---|---|---|---|
| ABS | FLOAT | PACK | SIN |
| ARCTAN | GET | PAGE | SQR |
| BOOLEAN | INPUT | PRED | SQRT |
| CHAR | INTEGER | PUT | SUCC |
| CHR | LN | READ | TEXT |
| COS | MAXINT | READLN | TRUE |
| DISPOSE | NEW | REAL | TRUNC |
| EOF | ODD | RESET | UNPACK |
| EOLN | ORD | REWRITE | WRITE |
| EXP | OUTPUT | ROUND | WRITELN |
| FALSE | | | |

Features at the extend and system levels add the following to the list of predeclared identifiers in MS-Pascal.

1. String intrinsics

|  |  |
|---|---|
| CONCAT | INSERT |
| COPYLST | POSITN |
| COPYSTR | SCANEQ |
| DELETE | SCANNE |

2. Extend level intrinsics

|  |  |
|---|---|
| ABORT | LOBYTE |
| BYWORD | LOWER |
| DECODE | RESULT |
| ENCODE | SIZEOF |
| EVAL | UPPER |
| HIBYTE |  |

3. System level intrinsics

|  |  |
|---|---|
| FILLC | MOVESL |
| FILLSC | MOVESR |
| MOVEL | RETYPE |
| MOVER |  |

4. Extend level I/O

|  |  |
|---|---|
| ASSIGN | READFN |
| CLOSE | READSET |
| DIRECT | SEEK |
| DISCARD | SEQUENTIAL |
| FCBFQQ | TERMINAL |
| FILEMODES |  |

5. INTEGER4 type

|  |  |
|---|---|
| BYLONG | LOWORD |
| FLOATLONG | MAXINT4 |
| HIWORD | ROUNDLONG |
| INTEGER4 | TRUNCLONG |

6. Super array type

LSTRING
NULL
STRING

7. WORD type

   MAXWORD
   WORD
   WRD

8. Miscellaneous

| | |
|---|---|
| ADRMEM | INTEGER2 |
| ADSMEM | REAL4 |
| BYTE | REAL8 |
| INTEGER1 | SINT |

# Chapter 5
# Introduction to Data Types

Types in Microsoft Pascal fall into three broad categories: simple, structured, and reference types. Table 5.1 that follows gives a breakdown of the types in each of these groups. The remainder of this chapter discusses types in general; Chapters 6 through 9 discuss the different groups in detail.

## 5.1 What Is a Type?

A type is the set of values that a variable or value can have within a program. Types are either predeclared or declared explicitly.

For example, the types INTEGER and REAL are predeclared, while the type ARRAY [1 . . 10] OF INTEGER is declared explicitly. An explicitly declared type may also be given a type identifier. To accomplish this latter task, a type declaration is required.

**Table 5.1**

**Categories of Types in Microsoft Pascal**

| Category | Includes | Comments/Examples |
|---|---|---|
| Simple Types | Original types | |
| | INTEGER | -MAXINT . . MAXINT |
| | WORD | 0 . . MAXWORD |
| | CHAR | CHR(0) . . CHR(255) |
| | BOOLEAN | (FALSE,TRUE) |
| | enumerated types | e.g., (RED,BLUE) |
| | subrange types | e.g., 100 . . 5000 |
| | REAL4, REAL8 | |
| | INTEGER4 | -MAXINT4 . . MAXINT4 |
| Structured Types | ARRAY OF type | |
| | general (OF any type) | |
| | SUPER ARRAY (OF type) | |
| | STRING (n) | [1 . . n] of CHAR |
| | LSTRING (n) | [0 . . n] of CHAR |
| | RECORD | |
| | SET OF type | |
| | FILE OF | |
| | general (binary) files | |
| | TEXT | Like FILE OF CHAR |
| Reference Types | Pointer Types | e.g., ^TREETIP |
| | ADR OF type | Relative address |
| | ADS OF type | Segmented address |
| Procedural and Functional Types | | Only as parameter type |

## 5.2 Declaring Data Types

The type declaration associates an identifier with a type of value. You declare types in the TYPE section of a program, procedure, function, module, interface, or implementation (not in the heading of a procedure or function).

A type declaration consists of an identifier followed by an equal sign and a type clause.

Examples of type definitions:

```
TYPE LINE  = STRING (80);
     PAGE = RECORD
            PAGENUM : 1 .. 499;
            LINES : ARRAY [1 .. 60] OF LINE;
            FACE : (LEFT, RIGHT);
            NEXTPAGE : ^PAGE
            END;
```

After declaring the data types, you declare variables of the types just defined in the VAR section of a program, procedure, function, module, or interface, or in the heading of a procedure or function. The following sample VAR section declares variables of the types in the preceding sample TYPE section:

```
VAR PARAGRAPH : LINE;
    BOOK : PAGE;
```

Because a type identifier is not defined until its declaration is processed by the compiler, a recursive type declaration such as the following is illegal:

```
T = ARRAY [0 .. 9] OF T;
```

Reference types require a standard exception to this rule and are discussed in Chapter 9, "Reference and Other Types."

A special feature of MS-Pascal is a category called super types. A super type declaration determines the set of types that designators of that super type can assume; it also associates an identifier with the super type. Super type declarations also occur in the TYPE section. The only super types currently available in MS-Pascal are super arrays.

## 5.3 Type Compatibility

MS-Pascal follows the ISO standard for type compatibility, with some additional rules added for super array types, LSTRINGs, and constant coercions (i.e., forced changes in the type of a constant). Type transfer functions, to override the typing rules, are available with some MS-Pascal features.

Two types can be "identical," "compatible," or "incompatible." An expression may or may not be "assignment compatible" with a variable, value parameter, or array index.

## 5.3.1 Type Identity and Reference Parameters

Two types are identical if they have the identical identifier or if the identifiers are declared equivalent with a type definition like the following:

```
TYPE T1 = T2;
```

"Identical" types are truly identical in MS-Pascal: there is no difference between types T1 and T2 in the example above. Type identity is based on the name of the types, and not on the way they are declared or structured. Thus, for example, T1 and T2 are not identical in the following declarations:

```
TYPE T1 = ARRAY [1 .. 10] OF CHAR;
     T2 = ARRAY [1 .. 10] OF CHAR;
```

Actual and formal reference parameters must be of identical types. Or, if a formal reference parameter is of a super array type, the actual parameter must be of the same super array type or a type derived from it. Two record or array types must be identical for assignment.

The only exception is for strings. Here, actual parameters of type CHAR, STRING, STRING (n), LSTRING, and LSTRING (n) are compatible with a formal parameter of super array type STRING. Also, the type of a string constant will change to any LSTRING type with a large enough bound. For example, the type of 'ABC' will change to LSTRING (5) if necessary.

Furthermore, an actual parameter of any FILE type may be passed to a formal parameter of a special record type FCBFQQ. Similarly, an actual parameter of type FCBFQQ may be passed to a formal parameter of any file type. See Section 8.7, "File I/O: System Level," for a description of the FCBFQQ type.

STRING (n) is a shorthand notation for:

```
    PACKED ARRAY [1 .. n] OF CHAR
```

The two types are identical. However, because variables with the type LSTRING are treated specially in assignments, comparisons, READs, and WRITEs, LSTRING (n) is not a shorthand notation for PACKED ARRAY [0 .. n] OF CHAR. The two types are not identical, compatible, or assignment compatible. See Section 7.2.3, "Using STRINGs and LSTRINGs," for further information on string types.

## 5.3.2 Type Compatibility and Expressions

Two simple or reference types are compatible if any of the following is true:

1. They are identical.
2. They are both ADR types.
3. They are both ADS types.
4. One is a subrange of the other.
5. They are subranges of compatible types.

Two structured types are compatible if any of the following is true:

1. They are identical.
2. They are SET types with compatible base types.
3. They are STRING derived types of equal length.
4. They are LSTRING derived types.

However, two structured types are incompatible if any of the following is true:

1. Either type is a FILE or contains a FILE.
2. Either type is a super array type.
3. One type is PACKED and the other is not.

Two values must be of compatible types when combined with an operator in an expression. (Most operators have additional limitations on the type of their operands. See Chapter 12, "Expressions," for details.) A CASE index expression type must be compatible with all CASE constant values.

## 5.3.3 Assignment Compatibility

Some types are implicitly compatible. This permits assignment across type boundaries. For instance, assume you declare the following variables:

```
VAR DESTINATION : T_DEST;
      SOURCE : T_SOURCE;
```

SOURCE is assignment compatible with DESTINATION (i.e., DESTINATION := SOURCE is permitted) if one of the following is true:

1. T_SOURCE and T_DEST are identical types.
2. T_SOURCE and T_DEST are compatible and SOURCE has a value in the range of subrange type T_DEST.
3. T_DEST is of type REAL and T_SOURCE is compatible with type INTEGER or INTEGER4.
4. T_DEST is of type INTEGER4 and T_SOURCE is compatible with type INTEGER or WORD.

Also, if T_DEST and T_SOURCE are compatible structured types, then SOURCE is assignment compatible with DESTINATION if one of the following is true:

1. For SETs, every member of SOURCE is in the base type of T_DEST.
2. For LSTRINGs, UPPER (DESTINATION) >= SOURCE.LEN.

Other than in the assignment statement itself, assignment compatibility is required in the following cases of implicit assignment:

1. passing value parameters
2. READ and READLN procedures
3. control variable and limits in a FOR statement
4. super array type array bounds, and array indices

Assignment compatibility is usually known at compile time, and an assignment generates simple instructions. However, some subrange, set, and LSTRING assignments depend on the value of the expression to be assigned and thus cannot be checked until runtime. If the range checking switch is on, assignment compatibility is checked at runtime; otherwise, no checking is done.

# Chapter 6
# Simple Types

The basic distinction between simple and structured data types is that simple types cannot be divided into other types, while structured types (discussed in Chapter 7, "Arrays, Records, and Sets," and Chapter 8, "Files") are composed of other types. The simple data types fall into three categories:

1. ordinal types
2. REAL
3. INTEGER4

## 6.1 Ordinal Types

Ordinal types are all finite and countable. They include the following simple types:

INTEGER

WORD

CHAR

BOOLEAN

enumerated types

subrange types

INTEGER4, though finite and countable, is not an ordinal type.

### 6.1.1 INTEGER

INTEGER values are a subset of the whole numbers and range from -MAXINT through 0 to MAXINT. MAXINT is the predeclared constant 32767 (i.e., 2^15 - 1) for current Microsoft Pascal target machines. (The value -32768 is not a valid INTEGER; the compiler uses it to check for uninitialized INTEGER and INTEGER subrange variables.)

INTEGER is not a subrange of INTEGER4 (discussed in Section 6.3, "INTEGER4"). If it were, signed expressions would have to be calculated using the INTEGER4 type and the result converted to INTEGER.

Expressions are always calculated using a base type, not a subrange type. INTEGER type constants may be changed internally to WORD type if necessary, but INTEGER variables cannot. INTEGER values change to REAL or INTEGER4 in an expression, if necessary, but not to WORD. The ORD function converts a value of any ordinal type to an INTEGER type.

The predeclared type INTEGER2 is identical to INTEGER.

## 6.1.2 WORD

The WORD and INTEGER types are similar, differing chiefly in their range of values. Both are ordinal types. You can think of WORD values as either a group of 16 bits or as a subset of the whole numbers from 0 to MAXWORD (65535, i.e., 2^16 - 1). The WORD type is an MS-Pascal feature that is useful in several ways:

1. to express values in the range from 32768 to 65535
2. to operate on machine addresses
3. to perform primitive machine operations, such as word ANDing and word shifting, without using the INTEGER type and running into the -32768 value

Unlike INTEGERs, all WORDs are nonnegative values. The WRD function changes any ordinal type value to WORD type. Like INTEGER values, WORD values in an expression are converted to the INTEGER4 type, if necessary.

Having both an INTEGER and a WORD type permits mapping of 16-bit quantities in either of two ways:

1. as a signed value ranging from -32767 to +32767
2. as a positive value ranging from 0 to 65535.

However, you must not mix WORD and INTEGER values in an expression (although doing so generates a warning rather than an error message). WORD and INTEGER values are not assignment compatible either.

### 6.1.3 CHAR

In MS-Pascal, CHAR values are 8-bit ASCII values. CHAR is an ordinal type. All 256-byte values are included in the type CHAR. In addition, SET OF CHAR is supported. Relational comparisons use the ASCII collating sequence.

Although the line-marker character used in TEXT files is not part of the CHAR type in the ISO standard, some target operating systems for MS-Pascal may require the line-marker character to be included (e.g., carriage return).

The CHR function changes any ordinal type value to CHAR type, as long as ORD of the value is in the range from 0 to 255. See Appendix D, "ASCII Character Codes," for a complete listing of the ASCII character set.

### 6.1.4 BOOLEAN

BOOLEAN is an ordinal type with only two (predeclared) values: FALSE and TRUE. The BOOLEAN type is a special case of an enumerated type, where ORD (FALSE) is 0 and ORD (TRUE) is 1. This means that FALSE < TRUE.

You may redefine the identifiers BOOLEAN, FALSE, and TRUE, but the compiler implicitly uses the ordinal (default) type in Boolean expressions and in IF, REPEAT, and WHILE statements.

No function exists for changing an ordinal type value to a BOOLEAN type value. However, you can achieve this effect with the ODD function for INTEGER and WORD values, or the expression:

```
    ORD (value) <> 0
```

### 6.1.5 Enumerated Types

An enumerated type defines an ordered set of values. These values are constants and are enumerated by the identifiers that denote them.

Examples of enumerated type declarations:

```
FLAGCOLOR = (RED, WHITE, BLUE)
SUITS = (CLUB, DIAMOND, HEART, SPADE)
DOGS = (MAUDE, EMILY, BRENDAN)
```

Every enumerated type is also an ordinal type. Identifiers for all enumerated type constants must be unique within their declaration level.

At the extend level, the READ and WRITE procedures and the ENCODE and DECODE functions operate on values of an enumerated type by treating the actual constant identifier as a string. This means that enumerated values can be read directly.

The ORD function, at the standard level, can be used to change enumerated values into INTEGER values; the WRD function changes enumerated values into WORD values.

The RETYPE function, at system level, can be used to change INTEGER or WORD values to an enumerated type. For example:

```
IF RETYPE (COLOR, I) = BLUE THEN WRITELN ('TRUE BLUE')
```

The values obtained by applying the ORD function to the constants of an enumerated type always begin with zero. Thus, the values obtained for the type FLAGCOLOR, from the example above, are as follows:

```
ORD (RED) = 0
ORD (WHITE) = 1
ORD (BLUE) = 2
```

Enumerated types are particularly useful for representing an abstract collection of names, such as names for operations or commands. Modifying a program by adding a new value to an enumerated type is much safer than using raw numbers, since any arrays indexed with the type or sets based on the type are changed automatically.

For example, interactive input of a command might be accomplished by reading the enumerated type identifier that corresponds to a command. Since enumerated types are ordered, comparisons like RED < GREEN may also be useful. At times, access to the lowest and highest values of the enumerated type is useful with the LOWER and UPPER functions, as in the following example:

```
VAR TINT : COLOR;
FOR TINT := LOWER (TINT) TO UPPER (TINT)
     DO PAINT (TINT)
```

### 6.1.6 Subrange Types

A subrange type is a subset of an ordinal type. The type from which the subset is taken is called the "host" type. Therefore, all subrange types are also ordinal types.

You can define a subrange type by giving the lower and upper bounds of the subrange (in that order). The lower bound must not be greater than the upper bound, but the bounds may be equal. The subrange type is frequently used as the index type of an array bound or as the base type of a set. See Chapter 7, "Arrays, Records, and Sets," for a discussion of arrays and sets.

Examples of subranges along with their host ordinal type:

| **Host Ordinal** | **Subrange** |
|---|---|
| INTEGER | 100 . . 200 |
| WORD | WRD(1) . . 9 |
| CHAR | 'A' . . 'Z' |
| enumerated type | RED . . YELLOW |

In addition, you may substitute a subrange clause for a list of values in the following circumstances:

1. when setting constants
2. when setting constructors
3. when setting CASE statement constants and record variant labels (at the extend level)

Besides using the subrange type in array and set declarations, you can use it to help guarantee that the value of a variable is within acceptable bounds. If the range checking switch is on during compilation, these bounds are checked at runtime.

For instance, if the logic of a program implies that a variable always has a value from 100 to 999, then declaring it with a subrange causes the compiler to check that the variable is never assigned a value outside this range.

In addition, declaring a subrange type may permit the compiler to allocate less room and use simpler operations. For example, declaring BOTTLES to be the INTEGER subrange 1 .. 100 means that the type can be allocated in eight bits instead of sixteen.

Three subrange types are predeclared:

1. BYTE = WRD(0) .. 255;
   {8-bit WORD subrange}
2. SINT = -127 .. 127;
   {8-bit INTEGER subrange}
3. INTEGER1 = SINT

The BYTE type is particularly useful in machine-oriented applications. For example, the ADRMEM and ADSMEM types (see Section 9.1.2, "Address Types," for details) normally treat memory as an array of bytes. However, since the BYTE type is really a subrange of the WORD type, expressions with BYTE values are calculated using 16-bit instead of 8-bit arithmetic, if necessary.

In some cases (for example, an assignment of a BYTE expression to a BYTE variable when the math checking switch is off), the compiler can optimize 16-bit arithmetic to 8-bit arithmetic. In general, using BYTE instead of WORD saves memory at the expense of BYTE-to-WORD conversions in expression calculations.

At the extend level, subrange bounds can be constant expressions. Because the compiler assumes that the left parenthesis always starts an enumerated type declaration, the first expression in a subrange declaration must not start with a left parenthesis. For example:

```
TYPE {First two are permitted.}
        FEE = (A, B, C);
        FIE = M + 2 * N . . (P - 2) * N;
        {FOO is invalid as declared.}
        FOO = (M + 2) * N . . P - 2 * N;
```

## 6.2 REAL

REAL values are nonordinal values of a given range and precision; the range of allowable values depends on the target system.

Most MS-Pascal implementations use either the Microsoft or IEEE single precision real number format. These formats have a 24-bit mantissa and an 8-bit exponent, giving about seven digits of precision and a maximum value of 1.701411E38. Microsoft format REAL constants are limited to the range 1.0E–38 to 1.0E+38. There is also a decimal real format controlled by the $decmath metacommand.

The current version of MS-Pascal includes expanded numeric data types for processing higher precision real (and integer) numbers. For reals, this includes support for single and double precision real numbers according to the IEEE floating-point standard, as well as single and double precision decimal-format numbers.

Standard Pascal provides a type REAL. MS-Pascal provides three real types: REAL, REAL4, and REAL8. However, the type REAL is always identical to either REAL4 or REAL8. The choice is made with a metacommand, $real:n, where n is either 4 or 8. {$real:8} has the same effect as TYPE REAL = REAL8. The default type for REAL is normally REAL4, but may be changed.

Any or all of these real number forms may be used in a single program. However, programs that use REAL4 and REAL8 will not be portable.

The REAL4 type is in 32-bit IEEE format, and the REAL8 type is in 64-bit IEEE format. The IEEE standard format is as follows:

| | |
|---|---|
| REAL4 | Sign bit, 8-bit binary exponent with bias of 127, 23-bit mantissa |
| REAL8 | Sign bit, 11-bit binary exponent with bias of 1023, 52-bit mantissa |

In both cases the mantissa has a "hidden" most significant bit (always one) and represents a number greater than or equal to 1.0 but less than 2.0. An exponent of zero means a value of zero, and the maximum exponent means a value called NAN ("Not A Number"). Bytes are in "reverse" order; the lowest addressed byte is the least significant mantissa byte.

The REAL4 numeric range is barely seven significant digits (24 bits), with an exponent range of E–38 to E+38. The REAL8 numeric range includes over fifteen significant digits (53 bits), with an exponent range of E–306 to E+306 (a very large number!).

The exponent character can be "D" or "d" as well as "E" or "e", so a number like 12.34d56 is permitted. This minor extension provides compatibility with other Microsoft languages. However, the D or d exponent character does not indicate double precision (as it does in FORTRAN), since this would imply that numbers with an E or e exponent characters are single precision.

REAL literals in MS-Pascal are converted first to REAL8 format and then to REAL4 as necessary (for example, to be passed as a CONST parameter or to initialize a variable in a VALUE section). If you need actual REAL4 constants, you must declare them as REAL4 variables (perhaps adding the READONLY attribute) and assign them a constant in a VALUE section.

Both REAL4 and REAL8 values are passed to intrinsic functions as reference (CONSTS) parameters, rather than as value parameters. The compiler accepts REAL expressions as CONSTS parameters; it will evaluate the expression, assign the result to a stack temporary, and pass the address of the temporary, which is usually more efficient than passing the value itself (especially in the REAL8 case).

Functions that return REAL values use the long return method; that is, the caller passes an additional, hidden, offset address of a stack temporary which will receive the result. This applies to all functions returning REAL4 or REAL8 values, both user-defined and intrinsic. See Section 12.2, "Boolean Expressions," for a description of REAL comparisons that produce an unordered result.

The MS-Pascal runtime libraries provide additional REAL functions to support Microsoft FORTRAN. These functions are available in MS-Pascal, but are not predeclared. See Chapter 15, "Available Procedures and Functions," for further information on the functions available and how to use them.

Base ten representation of REAL data is supported by Microsoft Pascal using a floating-point format. It consists of 6 (single) and 14 (double) binary coded decimal digits packed two to a byte. (If the exponent byte is zero, the number is zero.)

## 6.3 INTEGER4

Like INTEGER and WORD values, INTEGER4 values are a subset of the whole numbers. INTEGER4 values range from -MAXINT4 to MAXINT4. MAXINT4 is a predeclared constant with the value of 2,147,483,647 (i.e., 2^31 - 1). The value -2,147,487,648 (i.e., -2^31) is not a valid INTEGER4.

Unlike INTEGER and WORD, the INTEGER4 type is not considered an ordinal type. There are no INTEGER4 subranges and INTEGER4 cannot be an array index or the base type of a set. Also, INTEGER4 values cannot be used to control FOR and CASE statements.

INTEGER4 is currently an extended numeric type, like REAL. Values of type INTEGER or WORD in an expression change automatically to INTEGER4 if the expression requires an intermediate value that is out of the range of either INTEGER or WORD. Values of type INTEGER4 do not change to REAL in an expression; you must explicitly use the FLOATLONG function to make the conversion.

# Chapter 7
# Arrays, Records, and Sets

A structured type is composed of other types. The components of structured types are either simple types or other structured types. A structured type is characterized by the types of its components and by its structuring method. In Microsoft Pascal, a structured type can occupy up to 65534 bytes of memory.

The structured types in MS-Pascal are the following:

> ARRAY *range* OF *type*
>
> SUPER ARRAY *range* OF *type*
>
> > STRING (n)
> >
> > LSTRING (n)
>
> RECORD
>
> SET OF *base-type*
>
> FILE OF *type*

Because components of structures can be structured types themselves, you may have, for example, an array of arrays, a file of records containing sets, or a record containing a file and another record. This is an example of the data typing flexibility that provides Pascal with much of its linguistic power as a computing language.

The remainder of this chapter discusses arrays, records, and sets. See Chapter 8, "Files," for a discussion of files.

## 7.1 Arrays

An array type is a structure that consists of a fixed number of components. All of the components are of the same type (called the "component type").

The elements of the array are designated by indices, which are values of the "index type" of the array. The index type must be an ordinal type: BOOLEAN, CHAR, INTEGER, WORD, subrange, or enumerated.

Arrays in Pascal are one-dimensional, but since the component type can also be an array, n-dimensional arrays are supported as well.

Examples of type declarations for arrays:

```
TYPE
INT_ARRAY : ARRAY [1 .. 10] OF INTEGER;
ARRAY_2D : ARRAY [0 .. 7] OF ARRAY [0 .. 8] OF 0 .. 9;
MORAL_RAY : ARRAY [PEOPLE] OF (GOOD, EVIL)
```

In the last declaration, PEOPLE is a subrange type, while GOOD and EVIL are enumerated constants.

A short-hand notation available for n-dimensional arrays makes the following statement the same as the second example in the preceding paragraph:

```
ARRAY_2D : ARRAY [0 .. 7, 0 .. 8] OF 0 .. 9;
```

After declaring these arrays, you could assign to components of the arrays with statements such as these:

```
INT_ARRAY [10] := 1234;
ARRAY_2D [0,99] := 9;
MORAL_RAY [Machiavelli] := EVIL;
```

All of an n-dimensional PACKED array is packed; therefore these statements are equivalent:

```
PACKED ARRAY [1 .. 2, 3 .. 4] OF REAL

PACKED ARRAY [1 .. 2] OF PACKED ARRAY [3 .. 4] OF REAL
```

See Chapter 9, "Reference and Other Types," for a discussion of packed types.

## 7.2 Super Arrays

A super array is an example of an MS-Pascal "super type." A super type is like a set of types or like a function that returns a type. Super types in general, and super arrays in particular, are features of MS-Pascal.

The super array type has several important uses. You may use it for any of the following purposes:

1. To process strings

   Both STRING and LSTRING are predeclared super array types. The LSTRING type handles variable length strings. STRING handles fixed-length strings and strings more than 255 characters long.

2. To dynamically allocate arrays of varying sizes

   Otherwise such arrays would need a maximum possible size allocation.

3. As the formal parameter type in a procedure or function

   Such a declaration makes the procedure or function usable for a set or class of types, rather than for just a single fixed-length type.

A super type identifier specifies the set of types represented by the super type. A later type declaration may declare a normal type identifier as a type "derived" from that class of types. This derived type is like any other type.

A super array type declaration is an array type declaration prefixed with the keyword SUPER. Every array upper bound is replaced with an asterisk, as shown:

```
TYPE VECTOR = SUPER ARRAY [1 . . *] OF REAL;
```

Following the preceding type declaration, you could declare the following variables:

```
VAR ROW : VECTOR (10);
     COL : VECTOR (30);
     ROWP : ^VECTOR;
```

In this example, VECTOR is a super array type identifier. VECTOR (10) and VECTOR (30) are type designators denoting "derived types." ROW and COL are variables of types derived from VECTOR. ROWP is a pointer to the super array type VECTOR.

Although the general concept of super types allows other "types of types," such as super subranges and super sets (in addition to super arrays), super types currently allow only an array type with parametric upper bounds. A super type is a class of types and not a specific type. Thus, in the VAR section of a program, procedure, or function, you cannot declare the variables to be of a super type; you must declare them as variables of a type derived from the super type.

However, a formal reference parameter in a procedure or function can be given a super type: this allows the routine to operate on any of the possible derived types. (This kind of parameter is called a "conformant array" in other Pascals.)

A pointer referent type can also be given a super type. This allows a pointer to refer to any of the possible derived types. A pointer referent to a super type allows "dynamic arrays." These arrays are allocated on the heap by passing their upper bound to the procedure NEW. See Chapter 9, "Reference and Other Types," for a discussion of pointer types and dynamic allocation. See Chapter 15, "Available Procedures and Functions," for a description of the procedure NEW.

Example using the NEW procedure for dynamic allocation:

```
VAR STR_PNT : ^SUPER PACKED ARRAY [1 . . *] OF CHAR;
    VEC_PNT : ^SUPER ARRAY [0 . . *, 0 . . *] OF REAL;
      .
      .
    NEW (STR_PNT, 12 );
    NEW (VEC_PNT, 9, 99);
```

An actual parameter in a procedure or function can be of a super type rather than a derived type, but only if the parameter is a reference parameter or pointer referent. (These are the only kinds of variables that can be of a super rather than a derived type.)

Example of super arrays:

```
TYPE VECTOR = SUPER ARRAY [1 . . *] OF REAL;
{"VECTOR" is the super array type identifier.}

VAR X : VECTOR (12); Y : VECTOR (24); Z : VECTOR (36);
{X, Y, and Z are types derived from VECTOR.}
```

```
{Below, SUM accepts variables of all types}
{derived from the super type VECTOR.}
FUNCTION SUM (VAR V : VECTOR) : REAL;
{V is the formal reference parameter of the}
{super type VECTOR.}

VAR S : REAL; I : INTEGER;
BEGIN
   S := 0;
   FOR I := 1 TO UPPER (V) DO S := S + V [I];
   SUM := S
END;

BEGIN
   .
   .
   TOTAL := SUM (X) + SUM (Y) + SUM (Z);
   .
   .
END
```

The normal type rules for components of a super array type and for type designators that use a super array type allow components to be assigned, compared, and passed as parameters.

The UPPER function returns the actual upper bound of a super array parameter or referent. The maximum upper bound of a type derived from a super array type is limited to the maximum value of the index type implied by the lower bound (e.g., MAXINT, MAXWORD). Two super array types are predeclared, STRING and LSTRING. The compiler directly supports STRING and LSTRING types in the following ways:

1. LSTRING and STRING assignment
2. LSTRING and STRING comparison
3. LSTRING and STRING READs
4. access to the length of a STRING with the UPPER function
5. access to maximum length of an LSTRING with the UPPER function
6. access to LSTRING length with STR.LEN and STR[0]

These subjects are discussed in Section 7.2.3, "Using STRINGs and LSTRINGs."

## 7.2.1 STRINGs

STRINGs are predeclared super arrays of characters:

```
    TYPE STRING = SUPER PACKED ARRAY [1 . . *] OF CHAR;
```

A string literal such as 'abcdefg' automatically has the type STRING (n). The size of the array 'abcdefg' is 7; thus, the constant is of the STRING derived type, STRING (7).

Standard Pascal calls any packed array of characters with a lower bound of one a "string" and permits a few special operations on this type (such as comparison and writing) that you cannot do with other arrays.

In MS-Pascal, the super array notation STRING (n) is identical to PACKED ARRAY [1 . . n] OF CHAR (n may range from 1 to MAXINT). There is no default for n, as in some other Pascals, since STRING means the super array type itself and not a string with a default length.

The identifier STRING is for a super array, so you can only use it as a formal reference parameter type or pointer referent type. The other super array restrictions apply: you may not compare such a parameter or dereferenced pointer or assign it as a whole.

Any variable (or constant) with the super array type STRING, or one of the types CHAR or STRING (n) or PACKED ARRAY [1 . . n] OF CHAR, can be passed to a formal reference parameter of super array type STRING. Furthermore, a variable of type LSTRING or LSTRING (n) can also be passed to a formal reference parameter of type STRING. For a discussion of STRING as a formal reference parameter, see Section 7.2.3, "Using STRINGs and LSTRINGs."

Standard Pascal supports the assigning, comparing, and writing of STRINGs. The extend level permits reading STRINGs, including the super array type STRING and a derived type STRING (n). Reading a STRING causes input of characters until the end of a line or the end of the STRING is reached. If the end of the line is reached first, the rest of the STRING is filled with blanks. Writing a string writes all of its characters.

The customary Pascal type compatibility rules are relaxed for STRINGs. Any two variables or constants with the type PACKED ARRAY [1 . . n] OF CHAR or the type STRING (n) can be compared or assigned if the lengths are equal. However, since the length of a STRING super array type may vary, comparisons and assignments are not allowed.

Example of an illegal STRING assignment:

```
PROCEDURE CANNOT_DO (VAR S : STRING);
VAR STR : STRING (10);
BEGIN
   STR := S
   {This assignment is illegal because}
   {the length of S may vary.}
END;
```

The PACKED prefix in the declaration PACKED ARRAY [1 . . n] OF CHAR, as defined in the ISO standard, normally implies that a component cannot be passed as a reference parameter. In MS-Pascal, this restriction does not apply.

To keep conformance to the ISO standard, this passing of the CHAR component of a STRING as a reference parameter is defined as an "error not caught." Also, the index type of a string is officially INTEGER, but WORD type values can also be used to index a STRING. Many string-processing applications are expected to take advantage of the LSTRING type, described in Section 7.2.2, "LSTRINGs."

A number of intrinsic procedures and functions for strings are discussed in Chapter 15, "Available Procedures and Functions." Many of the procedures and functions described work on STRINGs; some apply only to LSTRINGs.

## 7.2.2 LSTRINGs

The LSTRING feature in MS-Pascal allows variable-length strings. LSTRING (n) is predeclared as:

```
TYPE LSTRING = SUPER PACKED ARRAY [0 . . *] OF CHAR
```

However, a variable with the explicit type PACKED ARRAY [0 . . n] OF CHAR is not "identical" to the type LSTRING (n) even though they are structurally the same. There is no default for n; the range of n is from zero to 255. Characters in an LSTRING can be accessed with the usual array notation.

Internally, LSTRINGs contain a length (L), followed by a string of characters. The length is contained in element zero of the LSTRING and can vary from 0 to the upper bound. The length of an LSTRING variable T can be accessed as T[0] with type CHAR, or as T.LEN with type BYTE. String constants of type CHAR or STRING (n) are changed automatically to type LSTRING.

The predeclared constant NULL is the empty string, LSTRING (0). NULL is the only constant with type LSTRING; there is no way to define other LSTRING constants. As with STRINGs, a CHAR component of an LSTRING can be passed as a reference parameter, and WORD and INTEGER values can be used to index an LSTRING.

Several operations work differently on LSTRINGs than on STRINGs. Any LSTRING can be assigned to any other LSTRING, so long as the current length of the right side is not greater than the maximum length of the left side. Similarly, an LSTRING can be passed as a value parameter to a procedure or function, so long as the current length of the actual parameter is not greater than the maximum length specified by the formal parameter. If the range checking switch is on, the compiler checks the assignment of LSTRINGs and the passing of LSTRING (n) parameters. The actual number of bytes assigned or passed is the minimum of the upper bounds of the LSTRINGs.

Neither side in an LSTRING assignment can be a parameter of the super array type LSTRING; both must be types derived from it.

Examples of LSTRING assignments:

```
{Declaring the variables}
VAR A : LSTRING (19);
      B : LSTRING (14);
      C : LSTRING (6);
        .
        .
{Assigning the variables}
A := '19 character string';
B := '14 characters';
```

```
C := 'shorty';
A := B;
{This is legal, since the length of B}
{is less than the maximum length of A.}
C := A;
{This is illegal, since length of A}
{is greater than the maximum length of C.}
```

You may compare any two LSTRINGs, including super array type LSTRINGs (the only super array type comparison allowed). Reading an LSTRING variable causes input of characters, until the end of the current line or the end of the LSTRING, and sets the length to the number of characters read. Writing from an LSTRING writes the current length string.

## 7.2.3 Using STRINGs and LSTRINGs

This section describes the STRING and LSTRING operations directly supported by the compiler. An annotated program at the end of this section illustrates the use of STRINGs and LSTRINGs in context.

See also Chapter 15, "Available Procedures and Functions," for descriptions of the following string procedures and functions:

| | |
|---|---|
| CONCAT | INSERT |
| COPYLST | POSITN |
| COPYSTR | SCANEQ |
| DELETE | SCANNE |

At the system level of MS-Pascal, the procedures FILLC, FILLSC, MOVEL, MOVESL, MOVER, and MOVESR also operate on strings.

MS-Pascal supports STRINGs and LSTRINGs directly in the following ways:

1. Assignment

   You may assign any LSTRING value to any LSTRING variable, as long as the maximum length of the target variable is greater than or equal to the current length of the source value and neither is the super array type LSTRING. If the maximum length of the target is less than the current length of the source, only the target length is assigned, and a runtime error occurs if the range checking switch is on. You may assign a STRING value to a STRING variable, as long as the length of both sides is the same and neither side is the super array type STRING. Passing either STRING or LSTRING as a value parameter is much like making an assignment.

2. Comparison

   The LSTRING operators < <= > >= <> = use the length byte for string comparisons; the operands may be of different lengths. Two strings must be the same length to be considered equal. If two strings of different lengths are equal up to the length of the shorter one, the shorter is considered less than the longer one. The operands can be of the super array type LSTRING. For STRINGs, the same relational operators are available, but the lengths must be the same and operands of the super array type STRINGs are not allowed.

3. READs and WRITEs

   READ LSTRING reads until the LSTRING is filled or until the end-of-line is found. The current length is set to the number of characters read. WRITE LSTRING uses the current length. See also READSET (described in Chapter 16, "File-Oriented Procedures and Functions"), which reads into an LSTRING as long as input characters are in a given SET OF CHAR. READ STRING pads with spaces if the line is shorter than the STRING. WRITE STRING writes all the characters in the string. Both READ and WRITE permit the super array types STRING and LSTRING, as well as their derived types.

4. Length access

   You can access the current length of an LSTRING variable T with T.LEN, which is of type BYTE, or with T[0], which is of type CHAR. This notation can assign a new length, as well as determine the current length. The UPPER function will find the maximum length of an LSTRING or the length of a STRING. This is especially useful for finding the upper bound of a super array reference parameter or pointer referent.

You cannot assign or compare mixed STRINGs and LSTRINGs, unless the STRING is constant. You can assign STRINGs to LSTRINGs, or vice versa, with one of the move routines or with the COPYSTR and COPYLST procedures. Since constants of type STRING or CHAR change automatically to type LSTRING if necessary, LSTRING constants are considered normal STRING constants. NULL (the zero length LSTRING) is the only explicit LSTRING constant.

In the sample program at the end of this section, all STRING parameters (CONST or VAR) may take either a STRING or an LSTRING; all LSTRING parameters are VAR LSTRING and must take an LSTRING variable.

A "special transformation" lets you pass an actual LSTRING parameter to a formal reference parameter of type STRING. The length of the formal STRING is the actual length of the LSTRING. Therefore, if LSTR (in the following example) is of type LSTRING (n) or LSTRING, it can be passed to a procedure or function with a formal reference parameter of type STRING:

```
VAR LSTR : LSTRING (10);
    .
    .
PROCEDURE TIE_STRING (VAR STR : STRING);
    .
    .
TIE_STRING (LSTR);
```

In this case, UPPER (STR) is equivalent to LSTR.LEN.

Procedures and functions with reference parameters of super type STRING can operate equally well on STRINGs and LSTRINGs. The only reason to declare a parameter of type LSTRING is when the length must be changed. Normally, an LSTRING is either a VAR or a VARS parameter in a procedure or function, since a CONST or CONSTS parameter of type LSTRING cannot be changed.

Example of a program that uses STRINGs and LSTRINGs:

```
PROGRAM STRING_SAMPLE;

PROCEDURE STRING_PROC (CONST S : STRING); BEGIN END;
PROCEDURE LSTRING_PROC (CONST S : LSTRING); BEGIN
END;

VAR
   CHR1VAR : CHAR;
   STR5VAR : STRING (5);
   LST5VAR : LSTRING (5);
   LST9VAR : LSTRING (9);
   STR4VAR : PACKED ARRAY [1 .. 4] OF CHAR;
   STR6VAR : PACKED ARRAY [1 .. 6] OF CHAR;

BEGIN

{Look at all the kinds of strings a}
{CONST STRING parameter takes.}
STRING_ PROC ('A');
{Character constant is OK.}
STRING_PROC (CHR1VAR);
{Character variable is OK.}
STRING_PROC ('STRING');
{STRING constant is OK.}
STRING_PROC (STR5VAR);
{STRING variable is OK.}
STRING_PROC (LST5VAR);
{LSTRING variable is OK.}

{However, a CONST LSTRING parameter cannot take}
{non-LSTRING variables.}
LSTRING_PROC ('A');
{Character constant is OK.}
LSTRING_PROC (CHR1VAR);
{Character variable is not OK!}
LSTRING_PROC ('STRING');
{STRING constant is OK.}
LSTRING_PROC (STR5VAR);
{STRING variable is not OK!}
LSTRING_PROC (LST5VAR);
{LSTRING variable is OK.}
```

```
{Assignments to a STRING variable are limited}
{to the same type.}
STR5VAR := 'A';
{Character constant is not OK!}
STR5VAR := CHR1VAR;
{Character variable is not OK!}
STR5VAR := 'TINY';
{STRING constant is too small.}
STR5VAR := 'RIGHT';
{Both sides have five characters; OK.}
STR5VAR := 'longer';
{Not OK; STRING constant is too large.}
STR5VAR := LST5VAR;
{Not OK; you cannot assign LSTRINGs to STRINGs.}
COPYSTR (LST5VAR, STR5VAR);
{COPYSTR is an intrinsic procedure.}
STR5VAR := STR4VAR;
{Not OK; STRING variable is too small.}
COPYSTR (STR4VAR, STR5VAR);
{COPYSTR is OK; padding of space in STR5VAR[5].}
STR5VAR := STR5VAR;
{OK; both sides have five characters.}
STR5VAR := STR6VAR;
{Not OK; STRING variable is too large.}

{Assignments to an LSTRING variable, however,}
{are more flexible.}
LST5VAR := 'A';
{Character constant is OK.}
LST5VAR := CHR1VAR;
{Character variable is not OK!}
LST5VAR := 'TINY';
{Smaller STRING constant is OK.}
LST5VAR := 'RIGHT';
{Same length STRING constant is OK.}
LST5VAR := 'LONGER';
{This gives an error at runtime only; OK for now.}
LST5VAR := LST9VAR;
{This may give an error at runtime; OK for now.}
LST9VAR := LST5VAR;
{This isn't even checked at runtime; always OK.}
LST5VAR := STR5VAR;
{Not OK; you cannot assign a STRING variable to an}
{LSTRING variable.}
COPYLST (STR5VAR, LST5VAR)
{This is the way to copy a STRING variable}
{to an LSTRING.}

END.
```

## 7.3 Records

A record structure acts as a template for conceptually related data of different types. The record type itself is a structure consisting of a fixed number of components, usually of different types.

Each component of a record type is called a field. The definition of a record type specifies the type and an identifier for each field within the record. Because the scope of these "field identifiers" is the record definition itself, they must be unique within the declaration. The field values associated with field identifiers are accessible with record notation or with the WITH statement.

For example, you could declare the following record type:

```
TYPE LP = RECORD
            TITLE : LSTRING (100);
            ARTIST : LSTRING (100);
            PLASTIC : ARRAY
               [1 . . SONG_NUMBER] OF SONG_TITLE
            END
```

You could then declare a variable of the type LP, as follows:

```
VAR BEATLES_1 : LP;
```

Finally, you could access a component of the record with either field notation or the WITH statement (note the period separating field identifiers):

```
BEATLES_1.TITLE := 'Meet The Beatles';
WITH BEATLES_1 DO
    PLASTIC[1] := 'I Wanna Hold Your Hand'
```

### 7.3.1 Variant Records

A record may have several "variants," in which case a certain field called the "tag field" indicates which variant to use. The tag field may or may not have an identifier and storage in the record. Some operations, such as the NEW and DISPOSE procedures and the SIZEOF function, can specify a tag value even if the tag is not stored as part of the record.

Examples of variant records:

```
TYPE OBJECT = RECORD
        X, Y : REAL;
        CASE S : SHAPE OF
            SQUARE : (SIZE, ANGLE : REAL);
            CIRCLE : (DIAMETER : REAL)
    END;

    FOO = RECORD
        CASE BOOLEAN OF
            TRUE : (I, J : INTEGER);
            FALSE : (CASE COLOR OF
                      BLUE : (X : REAL);
                      RED : (Y : INTEGER4))
    END;
```

Only one variant part per record is allowed; it must be the last field of the record. However, this variant part can also have a variant (and so on, to any level). All field identifiers in a given record type must be unique, even in different variants. For example, after declaring the record types above, you could create and then assign to the variables shown in the following example:

```
VAR O, P : OBJECT;
    F, G : FOO;

BEGIN
    O.DIAMETER := 12.34;        {CASE of CIRCLE}
    P.SIZE := 1.2;              {CASE of SQUARE}
    F.I := 1; F.J := 2;         {CASE of TRUE}
    G.X := 123.45;              {CASE of FALSE and BLUE}
    G.Y := 678999               {CASE of FALSE and RED;}
                                {this overwrites G.X.}
END;
```

The latest ISO standard requires every possible tag field value to select some variant. Therefore, it is illegal to include CASE INTEGER OF and omit a variant for every possible INTEGER value. However, such an omission is an error not caught in MS-Pascal.

MS-Pascal supports the use of full CASE constant options in the variant clause; that is, a list of constants can define a case. At the extend level, subranges and the OTHERWISE statement can also define a case. If used, OTHERWISE applies to the last variant in the list and is not followed by a colon. You can also declare an empty variant, such as POINT:( ) or OTHERWISE ( ). You can even declare an entirely empty record type, although the compiler issues a warning whenever the record is used.

The ISO standard defines a number of errors that relate to variant records; these errors may not be caught in MS-Pascal, even if the tag-checking switch is on. (The tag-checking switch generates code each time a variant field is used, to check that the tag value is correct.) In the record type declaration of OBJECT (in the previous example), any use of SIZE generates a check that S = SQUARE. However, in the case of FOO, uses of "I" cannot be checked because MS-Pascal does not allocate the BOOLEAN tag field.

The ISO standard further declares that when a "change of variant" occurs (such as when a new tag value is assigned), all the variant fields become undefined. However, MS-Pascal does not set the fields to an uninitialized value when a new tag is assigned. Therefore, using a variant field with an undefined value is an error not caught in MS-Pascal.

Nor does MS-Pascal enforce various restrictions on a record variable allocated on the heap with the long form of the NEW procedure. See Chapter 15, "Available Procedures and Functions," for details. However, MS-Pascal does check an assignment to such a "short record" to see that only the short record itself is modified in the heap.

A record allocated with the long form of NEW may be released using the short form of DISPOSE with no ill effects (this is an ISO error not caught in MS-Pascal). It is also an error not caught in MS-Pascal to DISPOSE of a record passed as a reference parameter or used by an active WITH statement.

Variant records interact with MS-Pascal features in two ways:

1. Declaring a variant that contains a file is not safe; any change to the file's data using a field in another variant may lead to I/O errors, even if the file is closed. In the following example, any use of R will lead to errors in F:

```
RECORD CASE BOOLEAN OF
       TRUE : (F : FILE OF REAL);
       FALSE : (R : ARRAY [1 . . 100] OF REAL)
       END;
```

2. Giving initial data to several overlapping variants in a variable in a VALUE section could have unpredictable results. In the following example, the initial value of LAP is uncertain:

   ```
   VAR LAP : RECORD CASE BOOLEAN OF
         TRUE : (I : INTEGER4);
         FALSE : (R : REAL)
         END;
   VALUE LAP.I := 10; LAP.R := 1.5;
   ```

MS-Pascal generates a warning message if you attempt either of these operations.

## 7.3.2 Explicit Field Offsets

MS-Pascal lets you assign explicit byte offsets to the fields in a record. This system level feature can be useful for interfacing to software in other languages, since control block formats may not conform to the usual MS-Pascal field allocation method. However, because it also permits unsafe operations, such as overlapping fields and word values at odd byte boundaries, it is not recommended unless the interface is necessary.

Example showing assignment of explicit byte offsets:

```
TYPE CPM =RECORD
                NDRIVE [00] : BYTE;
                FILENM [01] : STRING (8);
                FILEXT [09] : STRING (3);
                EXTENT [12] : BYTE;
                CPMRES [13] : STRING (20);
                RECNUM [33] : WORD;
                RECOVF [35] : BYTE
              END;

     OVERLAP = RECORD
                BYTEAR [00] : ARRAY [0 . . 7] OF BYTE;
                WORDAR [00] : ARRAY [0 . . 3] OF WORD;
                BITSAR [00] : SET OF 0 . . 63
              END;
```

As may be seen in the example, the offset is enclosed in brackets; this is similar to attribute notation. The number is the byte offset to the start of the field. Some target machines may not permit accessing a 16-bit value at an odd address, but the MS-Pascal Compiler doesn't catch this as an error.

If you give any field an offset, give offsets to all fields. For any offset that you omit, the compiler picks an arbitrary value. Although the compiler will process a declaration that includes both offsets and variant fields, you should use only one or the other in a given program.

Although you can completely control field overlap with explicit offsets, variants provide the long forms of the procedures NEW, DISPOSE, and SIZEOF. If you want to allocate different length records, use the RETYPE and GETHQQ procedures, instead of variants and the long form of NEW. For example:

```
CPMPV := RETYPE (CPMP, GETHQQ (36));
```

The compiler does support structured constants for record types with explicit offsets. Internally, odd length fields greater than one are rounded to the next even length. For example:

```
ODDR = RECORD
          F1[00] : STRING (3);
          F2[03] : CHAR
       END;
```

In this example, field F1 is four bytes long, so an assignment to F1 overwrites F2. In such a record, all odd length fields must be assigned first.

## 7.4 Sets

A set type defines the range of values that a set may assume. This range of assumable values is the "power set" of the base type you specify in the type definition. The power set is the set of all possible sets that could be composed from an ordinal base type. The null set, [ ], is a member of every set.

Suppose you declare the following set types:

```
TYPE HUES = SET OF COLOR;
     CAPS = SET OF 'A' .. 'Z';
     MATTER = SET OF (ANIMAL, VEGETABLE, MINERAL);
```

Then you declare variables like the following:

```
VAR FLAG : HUES;
    VOWELS : CAPS;
    LIVE : MATTER;
```

Finally, you could assign these set variables:

```
FLAG := [RED, WHITE, BLUE];
VOWELS := ['A', 'E', 'I', 'O', 'U'];
LIVE := [ANIMAL, VEGETABLE];
```

The set elements must be enclosed in brackets. This practice differs from the use of parentheses to enclose the base enumerated type in a set type declaration.

Set operations are implemented directly by generated in-line code or by routines in the set unit. See Chapter 12, "Expressions," for a complete discussion of operations on sets.

The ORD value of the base type can range from 0 to 255. Thus, SET OF CHAR is legal, but SET OF 1942 . . 1984 is not.

Sets whose maximum ORD value is 15 (i.e., sets that fit into a WORD) are usually more efficient than larger ones. Also, if the range checking switch is on, passing a set as a value parameter invokes a runtime compatibility check, unless the formal and actual sets have the same type.

Sets provide a clear and efficient way of giving several qualities or attributes to an object. In another language, you might assign each quality a power of two:

```
READY = 1
GETSET = 2
ACTIVE = 4
DONE = 8
```

You might then assign the qualities with a statement like this:

```
X := (READY + ACTIVE)
```

and then test them using OR and AND as bitwise operators with a statement like:

```
IF ((X AND ACTIVE) <> 0) THEN WRITELN ('GO FISH')
```

The equivalent declaration in MS-Pascal might be:

```
QUALITIES = SET OF (READY, GETSET, ACTIVE, DONE);
```

You could then assign the qualities with X := [GETSET, ACTIVE] and test them with the following operations:

| | |
|---|---|
| IN | tests a bit |
| + | sets a bit |
| - | clears a bit |

For example, an appropriate construction might be:

```
IF ACTIVE IN X THEN WRITELN ('GO FISH')
```

You can also use SET OF 0 . . 15 to test and set the bits in a WORD. Using WORDs both as a set of bits and as the WORD type requires giving two types to the word, with a variant record, the RETYPE function, or an address type.

The bits in a set are assigned starting with the most significant bit in the lowest addressed byte. Thus, on a byte-swapped machine, the set [0, 7, 8, 15] has the WORD value #80 + #01 + #8000 + #0100.

# Chapter 8 Files

A file is a structure that consists of a sequence of components, all of the same type. It is through files that Microsoft Pascal interfaces with a given operating system. Therefore, you must understand the FILE type in order to perform input to and output from a program.

## 8.1 Declaring Files

As with any other type, you must declare a file variable in order to use it. However, the number of components in a file is not fixed by declaring a FILE type.

Examples of FILE declarations:

```
TYPE F1 = FILE OF COLOR;
     F2 = FILE OF CHAR;
     F3 = TEXT;
```

Conceptually, a file is simply another data type, like an array, but with no bounds and with only one component accessible at a time. However, a file is usually associated with one of the following:

1. disk files
2. terminals
3. printers
4. other input and output devices

This implies the following restriction in MS-Pascal: a FILE OF FILE is illegal, directly or indirectly. Other structures, such as a FILE OF ARRAYs or an ARRAY OF FILEs, are permitted.

Most Pascal implementations connect file variables to the data files of the operating system. MS-Pascal always uses the target operating system to access files but does not impose additional formatting or structure on operating system files.

MS-Pascal supports normal statically allocated files, files as local variables (allocated on the stack), and files as pointer referents (allocated on the heap). Except for files in super arrays, the compiler generates code to initialize a file when it is allocated and to CLOSE a file when it is deallocated.

This initialization call occurs automatically in most cases. However, a file declared in a module or uninitialized unit's interface will only get its initialization call if you call the module or unit identifier as a procedure. File declarations in such cases get the following compiler warning:

```
    Contains file initialize module
```

Only a file in an interface of an uninitialized unit does not generate this warning.

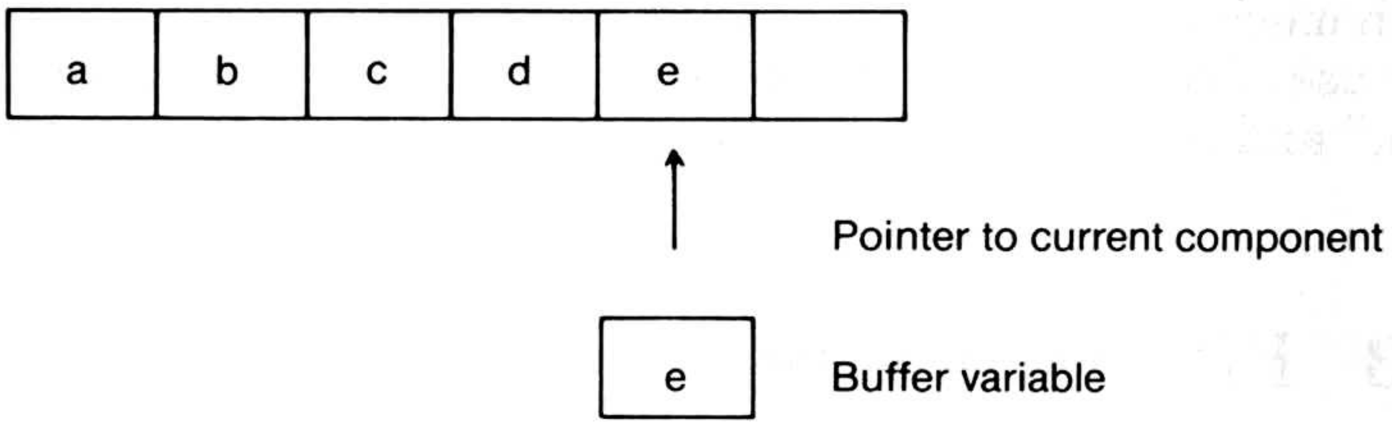
MS-Pascal sets up the standard files, INPUT and OUTPUT (discussed in Section 8.5, "The Predeclared Files: INPUT and OUTPUT"). In standard Pascal, files must be given in the program header, and when you run your program, the runtime system prompts you for filenames. At the extend level, you may use the ASSIGN and READFN procedures to give explicit operating system filenames to files not included in the program header.

Files in record variants or super array types are not recommended; if you use them, the compiler issues a warning. A file variable cannot be assigned, compared, or passed by value: it can only be declared and passed as a reference parameter.

At the extend level, you may indicate a file's access method or other characteristics by specifying the mode of the file. The mode is a value of the predeclared enumerated type FILEMODES. The modes available normally include the three base modes, SEQUENTIAL, TERMINAL, and DIRECT. All files, except INPUT and OUTPUT, are given SEQUENTIAL mode by default. INPUT and OUTPUT are given the default mode TERMINAL.

## 8.2 The Buffer Variable

Every file F has an associated buffer variable F^. A buffer variable and its associated file might look like this:



The procedures GET and PUT use this buffer variable to READ from and WRITE to files. GET copies the current component of the file to the buffer variable. PUT does the opposite; that is, PUT copies the value of the buffer variable to the current component.

The buffer variable can be referenced (i.e., its value fetched or stored) like any other MS-Pascal variable. This allows execution of assignments like the following:

```
F^ := 'z'
C := F^
```

A file buffer variable can be passed as a reference parameter to a procedure or function or used as a record in a WITH statement. However, the file buffer variable may not be updated correctly if the file position changes within the procedure, function, or WITH statement. The compiler issues a warning message to alert you to this possibility.

For example, the following use of a file buffer variable would generate a warning at compile time:

```
VAR A : TEXT;
PROCEDURE CHAR_PROC (VAR X : CHAR);
    .
    .
CHARPROC (A^);
{Warning issued here}
```

Two special internal mechanisms in MS-Pascal, lazy evaluation and concurrent I/O, allow, respectively, interactive terminal input in a natural way and overlapped I/O along with program execution. Lazy evaluation is applied to all ASCII structured files and is necessary for natural terminal input. Concurrent I/O is applied to all BINARY structured files and is necessary for some operating systems that support overlapping input and output.

Both mechanisms generate a runtime call that is executed before any use of the buffer variable. See Section 16.1.5, "Lazy Evaluation," and Section 16.1.6, "Concurrent I/O," for complete details.

## 8.3 File Structures

MS-Pascal files have two basic structures: BINARY and ASCII. These two structures correspond to raw data files and human-readable textfiles, respectively.

### 8.3.1 BINARY Structure Files

The Pascal data type FILE OF *type* corresponds to MS-Pascal BINARY structure files. These, in turn, correspond to unformatted operating system files.

Under operating systems that divide files into records, every record is one component of the file type (not to be confused with the record type). Primitive procedures such as GET and PUT operate on a record basis. Under operating systems that do not have their own record structure, the primitive procedures GET and PUT transfer a fixed number of bytes per call, equal to the length of one component. See Section 8.4, "File Access Modes," for further discussion of BINARY files.

### 8.3.2 ASCII Structure Files

The Pascal data type TEXT corresponds to MS-Pascal ASCII structure files. These, in turn, correspond to textual operating system files (called "textfiles" in this manual).

The Pascal TEXT type is like a FILE OF CHAR, except that groups of characters are organized into "lines" and, to a lesser extent, "pages." Primitive file procedures, such as GET and PUT, always operate on a character basis.

However, under operating systems that divide files into records, every record is a line (not a character). Even in operating systems that do not have their own record structure, other languages and utilities have some way of organizing bytes into lines of characters.

MS-Pascal provides a number of special functions and procedures that use this line-division feature. Because MS-Pascal does not impose any additional formatting on operating system files of most modes (including SEQUENTIAL, TERMINAL, and DIRECT), programs in other languages can generate and use these files.

MS-Pascal textfiles (files of type TEXT) are divided into lines with a "line marker," conceptually a character not of the type CHAR. In theory, a textfile can contain any value of type CHAR. However, under some operating systems, writing a particular character (say, CHR (13), carriage return, or CHR (10), line feed) may terminate the current line (record). This character value is the line marker in this case and, when read, always looks like a blank.

Under other operating systems, there may not be a terminating character. Still, as far as you are concerned, every line is followed by a line marker that reads as a blank.

At the extend level, a declaration for a textfile may include an optional line length. Setting the line length, which sets record length, is only needed for DIRECT mode textfiles. You may specify line length for other modes as well, but doing so has no effect.

Specify the line length of a textfile as a constant in parentheses after the word TEXT:

```
TYPE NAMEADDR = TEXT (128);
     DEFAULTX  = TEXT;
     SMALLBUF  = TEXT (2);
```

## 8.4 File Access Modes

The file modes in MS-Pascal are SEQUENTIAL, TERMINAL, and DIRECT. SEQUENTIAL and TERMINAL mode files are available at the standard level; all three, including DIRECT mode, are available at the extend level. SEQUENTIAL and TERMINAL mode ASCII structure files can have variable length records (lines); DIRECT mode files must have fixed length records or lines.

The declaration of a file in Pascal implies its structure, but not its mode. For example, FILE OF STRING (80) indicates BINARY structure; TEXT indicates ASCII structure. An assignment like F.MODE := DIRECT sets the mode; this only works at the extend level and is currently only needed to set DIRECT mode.

## 8.4.1 TERMINAL Mode Files

TERMINAL mode files always correspond to an interactive terminal or printer. TERMINAL mode files, like SEQUENTIAL mode files, are opened at the beginning of the file for either reading or writing. Records are accessed one after the other until the end of the file is reached.

Operation of TERMINAL mode input for terminals depends on the file structure (ASCII or BINARY). For ASCII structure (type TEXT), entire lines are read at one time. This permits the usual operating system intraline editing, including backspace, advance cursor, and cancel. Characters are echoed to the terminal screen while the line is being typed.

If the target operating system does not support intraline editing or echo, the MS-Pascal file system interface provides it. However, since an entire line is read at once, you cannot read the characters as you type them, invoke several prompts and responses on the same line, and so on.

For BINARY structure TERMINAL mode (usually type FILE OF CHAR), you can read characters as you type them. No intraline editing or echoing is done. This method permits screen editing, menu selection, and other interactive programming on a keystroke rather than line basis.

TERMINAL mode files use lazy evaluation to properly handle normal interactive reading of the terminal keyboard. See Section 16.1.5, "Lazy Evaluation," for details.

### 8.4.2 SEQUENTIAL Mode Files

SEQUENTIAL mode files are generally disk files or sequential access devices. Like TERMINAL mode files, SEQUENTIAL mode files are opened at the beginning of the file for either reading or writing, and records are accessed one after another until the end of the file. Standard Pascal files are in SEQUENTIAL mode by default (except for INPUT and OUTPUT).

### 8.4.3 DIRECT Mode Files

DIRECT mode files are generally disk files or other random access devices. DIRECT mode files and the ability to specify the access mode for file I/O are available options at the extend level of MS-Pascal.

DIRECT mode ASCII structure files, as well as all BINARY structure files, have fixed-length records, where a record is either a line or file component. (Here the term "record" refers not to the normal Pascal record type, but to a disk structuring unit.) DIRECT files are always opened for both reading and writing, and records can be accessed randomly by record number. There is no record number zero; records begin with record number one.

## 8.5 The Predeclared Files: INPUT and OUTPUT

Two files, INPUT and OUTPUT, are predeclared in every MS-Pascal program. These files get special treatment as program parameters and are normally required as parameters in the program heading:

```
    PROGRAM ACTION (INPUT, OUTPUT);
```

If there are no program parameters and the program does not use the files INPUT and OUTPUT, the heading can look like this:

```
    PROGRAM ACTION;
```

However, you should include INPUT and OUTPUT as program parameters if you use them, either explicitly or implicitly, in the program itself:

```
WRITE (OUTPUT, 'Prompt: ')      {Explicit use}
WRITE ('Prompt: ')              {Implicit use}
```

These examples would generate a warning if OUTPUT was not declared in the program heading. The only effect of INPUT and OUTPUT as program parameters is to suppress this warning.

Although you may redefine the identifiers INPUT and OUTPUT, the file assumed by textfile input and output procedures and functions (e.g., READ, EOLN) is the predeclared definition. The procedures RESET (INPUT) and REWRITE (OUTPUT) are generated automatically, whether or not INPUT and OUTPUT are present as program parameters (you may also use these procedures explicitly).

INPUT and OUTPUT have ASCII structure and TERMINAL mode. They are initially connected to your terminal and opened automatically. At the extend level of MS-Pascal, you can change these characteristics if you wish.

## 8.6 File I/O: Extend Level

A file variable in MS-Pascal is really a record, of type FCBFQQ, called a file control block. At the extend level, a few standard fields in this record help you handle file modes and error trapping.

Additional fields and the record type FCBFQQ itself can be used at the system level, described in Section 8.7, "File I/O: System Level." Along with access to certain FCB fields, extend level I/O also includes the following procedures:

| | |
|---|---|
| ASSIGN | READFN |
| CLOSE | READSET |
| DISCARD | SEEK |

See Section 16.3, "Extend Level I/O," for a description of these procedures.

Use the normal record field syntax to access FCB fields. For a file F, the fields are named F.MODE, F.TRAP, and F.ERRS. You may change or examine these fields at any time.

1. F.MODE := FILEMODES;

   This field contains the mode of the file: SEQUENTIAL, TERMINAL, or DIRECT. These values are constants of the predeclared enumerated type FILEMODES. The file system uses the MODE field only during RESET and REWRITE. Thus, changing the MODE field of an open file has no effect and is, in fact, discouraged. Except for INPUT and OUTPUT, which have TERMINAL mode, a file's mode is SEQUENTIAL by default.

   RESET and REWRITE change the mode from SEQUENTIAL to TERMINAL if they discover that the device being opened is a terminal or printer and if the target operating system allows it. This is useful in programs designed to work either interactively or in batch mode. You must set DIRECT mode before RESET or REWRITE if you plan to use SEEK on a file.

2. F.TRAP := BOOLEAN;

   If this field is TRUE, error trapping for file F is turned on. Then, if an input/output error occurs, the program does not abort and the error code can be examined. Initially, F.TRAP is set FALSE. If FALSE and an I/O error occurs, the program aborts.

3. F.ERRS := WRD(0) . . 15;

   This field contains the error code for file F. An error code of zero means no error; values from 1 to 15 imply an error condition. If you attempt a file operation other than CLOSE or DISCARD and F.ERRS is not zero, the attempted file operation is ignored and the program continues. After the error is trapped, the program must set F.ERRS back to zero to prevent succeeding file operations from being ignored.

   CLOSE and DISCARD do not examine the initial value of F.ERRS, so they are never ignored and do not cause an immediate abort. Nevertheless, if CLOSE or DISCARD themselves generate an error condition, F.TRAP is used to determine whether to trap the error or to abort.

An operation ignored because of an error condition does not change the file itself, but may change the buffer variable or READ procedure input variables. See Appendix H, "Messages," for a complete listing of error messages and warnings.

Also at the extend level, you may set the line length for a textfile, as shown:

```
TYPE SMALLBUF = TEXT (16);
VAR RANDOMTEXT : TEXT (132);
```

Declaring line length applies only to DIRECT mode ASCII structure files, where the line length is the record length used for reading and writing. Setting the line length has no effect on other ASCII files.

## 8.7 File I/O: System Level

At the system level of MS-Pascal, you can call procedures and functions that have a formal reference parameter of type FCBFQQ with an actual parameter of the type FILE OF *type* or TEXT, or the identical FCBFQQ type.

The FCBFQQ type is the underlying record type used to implement the file type in MS-Pascal. The interface for the target system FCBFQQ type (and any other types needed) is usually part of the internal file system. Thus, procedures and functions that reference FCBFQQ parameters can be called with any file type, including predeclared procedures and functions like CLOSE and READ.

An FCBFQQ type variable can be passed to procedures like READLN and WRITELN that require a textfile. This permits, for example, calling directly the interface routines on the target operating system, working with mixtures of MS-Pascal and MS-FORTRAN (which share the file system interface but have special FCBFQQ fields), and other special file system activities.

Such activities require a sound knowledge of the file system.

# Chapter 9
# Reference and Other Types

The array, record, and set types discussed in Chapter 7, "Arrays, Records, and Sets," let you describe data structures whose form and size are predetermined and whose components are accessed in a standard way. The file type, described in Chapter 8, "Files," is a structure that varies in size but whose form and means of access are predetermined.

In this chapter, you will find a discussion of reference types, which allow data structures that vary in size and form and whose means of access is particular to the programming problem involved. Also included are notes on PACKED types and procedural and functional types.

## 9.1 Reference Types

A reference to a variable or constant is an indirect way to access it. The pointer type is an abstract type for creating, using, and destroying variables allocated from an area called the heap. The heap is a dynamically growing and shrinking region of memory allocated for pointer variables.

Microsoft Pascal also provides two machine-oriented address types: one for addresses that can be represented in 16 bits, the other for addresses that require 32 bits.

Pointers are generally used for trees, graphs, and list processing. Use of pointers is portable, structured, and relatively safe.

Address types provide an interface to the hardware and operating system; their use is frequently unstructured, machine specific, low level, and unsafe. Both pointers and address types are discussed further in the following sections.

### 9.1.1 Pointer Types

A pointer type is a set of values that point to variables of a given type. The type of the variables pointed to is called the "reference type." Reference variables are all dynamically allocated from the heap with the NEW procedure. Pascal variables are normally allocated on the stack or at fixed locations.

You may perform only the following actions on pointers:

1. assign them
2. test them for equality and inequality with the two operators = and <>
3. pass them as value or reference parameters
4. dereference them with the up arrow (^)

Every pointer type includes the pointer value NIL. Pointers are frequently used to create list structures of records, as shown in the following example:

```
TYPE
   TREETIP = ^TREE;
   TREE = RECORD
               VAL : INTEGER;
                    {Value of TREE cell.}
               LEFT, RIGHT : TREETIP
                    {Pointers to other TREETIP cells.}
                    {Note recursive definition.}
            END;
```

Unlike most type declarations, the declaration for a pointer type can refer to a type of which it is itself a component. The declaration can also refer to a type declared later in the same TYPE section, as in TREE and TREETIP in the previous example.

Such a declaration is called a forward pointer declaration and permits recursive and mutually recursive structures. Because pointers are so often used in list structures, forward pointer declarations occur frequently.

The compiler checks for one ambiguous pointer declaration. Suppose the previous example was in a procedure nested in another procedure that also declared a type TREE. Then the reference type of TREETIP could be either the outer definition or the one following in the same TYPE section. MS-Pascal assumes the TREE type intended is the one later in the same TYPE section and gives the warning:

```
Pointer Type Assumed Forward
```

At the extend level, a pointer can have a super array type as a referent type. The actual upper bounds of the array are passed to the NEW procedure to create a heap variable of the correct size. Forward pointer declarations of the super array type are not allowed.

MS-Pascal conforms to the ISO requirement for strict compatibility between pointers. For example, you cannot declare two pointers with different types and then assign or compare them, even if they happen to point to the same underlying type. For example:

```
VAR PRA : ^REAL;
      PRE : ^REAL;
BEGIN PRA := PRE {This is illegal!}
END;
```

Programs usually contain only one type declaration for a pointer to a given type. In the TREETIP example, the type of LEFT and RIGHT could be ^TREE instead of TREETIP, but then you couldn't assign variables of type TREETIP to these fields. However, it is sometimes useful to make sure that two classes of pointers are not used together, even if they point to the same type.

For example, suppose you have a type RESOURCE kept in a list and declare two types, OWNER and USER, of type ^RESOURCE. The compiler would catch assignment of OWNER values to USER variables and vice versa and issue a warning message.

In theory, pointers have nothing to do with actual machine addresses. In fact, a pointer may be implemented in different ways on different target machines. A pointer may be implemented as a normal address, as a segment offset address, as an offset from one or more fixed locations, or as an indirect address, among other possibilities.

If the initialization checking switch is on, a newly created pointer has an uninitialized value. If the NIL checking switch is on, pointer values are tested for various invalid values. Invalid values include NIL, uninitialized values, reference to a heap item that has been DISPOSEd, or a value that is not valid as a heap reference.

## 9.1.2 Address Types

As a system implementation language, MS-Pascal needs a method of creating, manipulating, and dereferencing actual machine addresses. The pointer type is only applicable to variables in the heap.

There are two kinds of addresses: relative and segmented. The keywords ADR and ADS refer to the relative address type and the segmented address type, respectively. As the following example shows, you use the keywords both as type clause prefixes and as prefix operators:

```
VAR INT_VAR : INTEGER;
      REAL_VAR : REAL;
      A_INT : ADR OF INTEGER;
      {Declaration of ADR variable}
      AS_REAL : ADS OF REAL;
      {Declaration of ADS variable}

BEGIN
   INT_VAR := 1;
   {Normal integer variable}
   REAL_VAR := 3.1415;
   {Normal real variable}
   A_INT := ADR INT_VAR;
   {ADR used as operator}
   AS_REAL := ADS REAL_VAR;
   {ADS used as operator}
   WRITELN (A_INT^, AS_REAL^)
   {Note use of up arrow to dereference}
   {the address types.}
   {Output is 1 and 3.1415.}
END.
```

The characteristics of relative and segmented address types, as implemented for different machines, are shown in Table 9.1.

**Table 9.1**

**Relative and Segmented Machine Addresses**

| Machine | ADR | ADS |
|---|---|---|
| 8080 | 16-bit absolute | Same as ADR |
| 8086 | 16-bit default data segment offset | 16-bit offset, 16-bit segment |
| Z8000 (unsegmented) | 16-bit data absolute | Same as ADR |
| Z8000 (segmented) | Same as ADS | 16-bit segment, 16-bit offset |

See your *Microsoft Pascal Compiler User's Guide* for details specific to your implementation of the compiler.

In MS-Pascal, you may declare a variable that is an address:

```
VAR X : ADR OF BYTE;
```

Then, with the following record notation, you can assign numeric values to the actual variable:

```
X.R := 16#FFFF
```

In an unsegmented environment, the .R (relative address) is the only record field available for ADR and ADS addresses.

Since MS-Pascal allows nondecimal numbering, you may specify the assigned value in hexadecimal notation. You may also assign to a segment field with the ADS type in a segmented environment, using the field notation .S (segment address). Thus, you may declare a variable of an ADS type and then assign values to its two fields:

```
VAR Y : ADS OF WORD;
    .
    .
Y.S := 16#0001
Y.R := 16#FFFF
```

As shown above, any 16-bit value can be directly assigned to address type variables, using the .R and .S fields. The ADR and ADS operators obtain these addresses directly. The example below assigns addresses this way to the variables X and Y:

```
VAR X  : ADR OF BYTE;
    Y  : ADS OF WORD;
    W  : WORD;
    B  : BYTE;
         .
         .
    X  := ADR B;
    Y  := ADS W;
```

MS-Pascal supports these two predeclared address types:

```
ADRMEM = ADR OF ARRAY [0 . . 32766] OF BYTE;
ADSMEM = ADS OF ARRAY [0 . . 32766] OF BYTE;
```

Since the type referred to by the address is an array of bytes, byte indexing is possible. For example, if A is of type ADRMEM, then A^[15] is the byte at the address A.R + 15, where .R specifies an actual 16-bit address.

You can use the address types for a constant address (a form of structured constant); you may also take the address of a constant or expression. For example:

```
TYPE ADRWORD = ADR OF WORD;
      ADSWORD = ADS OF WORD;
VAR W : WORD;
    R  : ADRWORD;
CONST CONADR = ADRWORD (1234);
BEGIN
   W := CONADR^;
   {Get word at address 1234}
   W := ADSWORD (0, 32)^;
   {Get word at address 0:32}
   W := (ADS W).S;
   {Get value of DS segment register}
   R := ADR '123';
   {Get address of a constant value}
   R := ADR (W DIV 2 + 1)
   {Get address of expression value}
END;
```

However, constants or expressions that yield addresses cannot currently be used as the target of an assignment (or as a reference parameter or WITH record), as shown:

```
CONST ADSCON = ADSWORD (256, 64);       {OK}
FUNCTION SOME_ADDRESS: ADSWORD;  {OK}
BEGIN
   ADSWORD (0, 32)^ := W; {Not permitted}
   ADSCON^ := 12; {Not permitted}
   SOME_ADDRESS^ := 100 {Not permitted}
END;
```

## 9.1.3 Segment Parameters for the Address Types

Two keywords, VARS and CONSTS, are available as parameter prefixes, like VAR and CONST, to pass the segmented address of a variable. If P is of type ADS FOO, then P^ can be passed to a VARS formal parameter, such as VARS X : FOO, but cannot be passed to a VAR formal parameter.

In a segmented machine environment, a default data segment is assumed, in which case a VAR parameter is passed as the default data segment offset of a variable. A VARS parameter is passed as both the segment value and the offset value.

In the 8086 environment, both VARS parameters and ADS variables have the offset (.R) value in the WORD with the lower address and the segment (.S) value in the address plus two.

In the segmented Z8000 environment, the segment (.S) value is in the lower address and the offset (.R) value in the address plus two. Also, the ADR type is identical to the ADS type.

In the nonsegmented environment (e.g., 8080), VAR and CONST are identical to VARS and CONSTS. Since ADS and ADR are identical in a nonsegmented environment, the ADS type is useful in situations where the target environment may change. For example, in MS-Pascal, some primitive file system calls are declared with ADS parameters.

In pointer type declarations, the up arrow (^) prefixes the type pointed to; in program statements, it dereferences a pointer so that the value pointed to can be assigned or operated on. The up arrow also dereferences ADR and ADS types in program statements.

Component selection with the up arrow (^) is performed before the unary operators ADR or ADS. Because the up arrow (^) selector can appear after any address variable to produce a new variable, it can occur, for example, in the target of an assignment, a reference parameter, as well as in expressions. Since ADS and ADR are prefix operators, they are used only in expressions, where they apply only to a variable or constant or expression.

Pascal is a strongly typed language; two pointer variables are compatible only if they have the same type (it is not enough that they point to the same type). However, two address types are considered the same type if they are both ADR or both ADS types. This lets you assign an ADR OF WORD to an ADR OF STRING (200). Such an assignment would make it easy to wipe out part of memory by assigning a variable of type STRING (200) to the 200 bytes starting at the address of a WORD variable.

If P1 is type ADR OF STRING (200) and P2 is any ADR OF type, the assignment P1^ := P2^ generates fast code with no range checking. Although this capability is not safe, operating systems and other software sometimes require it.

ADR and ADS are not compatible with each other, but the .R notation should overcome or reduce the problem.

## 9.1.4 Using the Address Types

Within limits, you may combine and intermingle the two address types. The following example illustrates the rules that apply in a segmented environment:

```
VAR
   P : ADS OF DATA;
   {P is segmented address of type DATA.}
   Q : ADR OF DATA;
   {Q is relative address of type DATA.}
   X : DATA;
   {X is some variable of type DATA.}
```

```
BEGIN
   P := ADS X;
   {Assign the address of X to P.}
   X := P^;
    {Assign to X the value pointed to by P.}
   P := ADS P^;
    {Assign to P the address of the value whose}
   {address is pointed to by P. P is unchanged}
   {by this assignment.}
   Q := ADR X;
   {Assign the relative address of X to Q.}
   Q.R := (ADR X).R;
   {Assign the relative address of X to Q,}
   {using the WORD type.}
   P := ADS Q^;
    {Assign the address of the variable at Q to P.}
   {You can always apply ADS to ADR^.}
   Q := ADR P^;
    {Illegal; you cannot apply ADR to ADS ^.}
   P.R := 16#8000;
   {Assign 32768 to P's offset field.}
   P.S := 16;
   {Assign 16 to P's segment field.}
   Q.R := P.R + 4
   {Assign P's offset plus 4 to be the value of Q.}
END;
```

See also the examples given in Section 9.1.2, "Address Types."

### 9.1.5 Notes on Reference Types

The address type and pointer type should be treated as two distinct types. The pointer type, in theory, is just an undefined mapping from one variable to another variable. The method of implementation is undefined. However, the address type deals with actual machine addresses.

Therefore, the pointer type is an abstract data type that works the same in all implementations; the address type is generally not portable, unless used with some caution. Address types are portable only if you restrict yourself to using ADS and never assign to fields. Even with these restrictions, however, they can be quite useful.

The following special facilities that use pointer variables are not allowed with address variables.

1. The NEW and DISPOSE procedures are only permitted with pointers. NIL does not apply to the address type. There are no special address values for empty, uninitialized, or invalid addresses.
2. The type "address of super array type" is not supported in the same way as "pointer to super array type." Getting the address of a super array variable is still permitted with ADR and ADS. For example, if a procedure or function formal parameter is declared as VAR S : STRING, then within the procedure or function, the expression ADS S is fine. Unlike a pointer, the address does not contain any upper bounds.

## 9.2 PACKED Types

Any of the structured types can be PACKED. This could economize storage at the possible expense of access time or access code space. However, in MS-Pascal, some limitations on the use of PACKED structures currently apply:

1. The prefix PACKED is always ignored, except for type checking, in sets, files, and arrays of characters. In most versions of MS-Pascal it has no actual effect on the representation of records and other arrays. Furthermore, PACKED can only precede one of the structure names ARRAY, RECORD, SET, or FILE; it cannot precede a type identifier. For example, if COLORMAP is the identifier for an unpacked array type, "PACKED COLORMAP" is not accepted.
2. A component of a PACKED structure cannot be passed as a reference parameter or used as the record of a WITH statement, unless the structure is of a string type. Also, obtaining the address of a PACKED component with ADR or ADS is not permitted.
3. A PACKED prefix only applies to the structure being defined: any components of that structure that are also structures are not packed unless you explicitly include the reserved word PACKED in their definition. The only exception to this rule, n-dimensional arrays, is discussed in Section 7.1, "Arrays."

## 9.3 Procedural and Functional Types

Procedural and functional types are different from other MS-Pascal types. (Wherever the term "procedural" is used from here on, both procedural and functional is implied.) You may not declare an identifier for a procedural type in a TYPE section; nor may you declare a variable of a procedural type. However, you may use procedural types to declare the type of a procedural parameter, and in this sense they conform to the Pascal idea of a type.

A procedural type defines a procedure or function heading and gives any parameters. For a function, it also defines the result type. The syntax of a procedural type is the same as a procedure or function heading, including any attributes. There are no procedural variables in MS-Pascal, only procedural parameters.

Example of a procedural type declaration:

```
    PROCEDURE ZERO (FUNCTION FUN (X, Y : REAL) : REAL)
```

The parameter identifiers in a procedural type (X and Y in the previous example) are ignored; only their type is important.

See Section 14.4.3, "Procedural and Functional Parameters," for more information about procedural types in MS-Pascal.

# Chapter 10
# Constants

## 10.1 What Is a Constant?

A constant is a value that is known before a program starts and that will not change as the program progresses. Examples of constants include the number of days in the week, your birthdate, the name of your dog, or the phases of the moon.

A constant may be given an identifier, but you cannot alter the value associated with that identifier during the execution of the program. When you declare a constant, its identifier becomes a synonym for the constant itself.

Each constant implicitly belongs to some category of data:

1. Numeric constants (discussed in Section 10.3, "Numeric Constants") are one of the several number types: REAL, INTEGER, WORD, or INTEGER4.
2. Character constants (discussed in Section 10.4, "Character Strings") are strings of characters enclosed in single quotation marks and are called "string literals" in Microsoft Pascal.
3. Available at the extend level, structured constants (discussed in Section 10.5, "Structured Constants") include constant arrays, records, and typed sets.

Also available at the extend level, constant expressions (discussed in Section 10.6, "Constant Expressions") let you compute a constant based on the values of previously declared constants in expressions.

The identifiers defined in an enumerated type are constants of that type and cannot be used directly with numeric (or string) constant expressions. These identifiers can be used with the ORD, WRD, or CHR functions (e.g., ORD (BLUE)). The extend level also permits directly reading and writing the enumerated type's constant identifiers as character strings.

TRUE and FALSE are predeclared constants of type BOOLEAN and can be redeclared. NIL is a constant of any pointer type; however, because it is a reserved word, you may not redefine it. Also, the null set is a constant of any set type.

Numeric statement labels have nothing to do with numeric constants; you may not use a constant identifier or expression as a label. Internally, all constants are limited in length to a maximum of 255 bytes.

## 10.2 Declaring Constant Identifiers

Declaring a constant identifier introduces the identifier as a synonym for the constant. You put these declarations in the CONST section of a compiland, procedure, or function.

The general form of a constant identifier declaration is the identifier followed by an equal sign and the constant value. The following program fragment includes three statements that identify constants (beginning after the word "CONST"):

```
PROGRAM DEMO (INPUT, OUTPUT);
CONST DAYSINYEAR = 365;
        DAYSINWEEK = 7;
        NAMEOFPLANET = 'EARTH';
```

In this example, the numbers 365 and 7 are numeric constants; 'EARTH' is a string literal constant and must be enclosed in single quotation marks.

When you compile a program, the constant identifiers are not actually defined until after the declarations are processed. Thus, a constant declaration like the following has no meaning:

```
N = -N
```

The ISO standard defines a strict order in which to set out the declarations in the declaration section of a program:

```
CONST MAX = 10;
TYPE NAME = PACKED ARRAY [1 . . MAX] OF CHAR;
VAR FIRST : NAME;
```

MS-Pascal relaxes this order and, in fact, allows more than one instance of each kind of declaration:

```
TYPE COMPLEX = RECORD R, I : REAL END;
CONST PII = COMPLEX (3.1416, 00 );
VAR PIX : COMPLEX;
TYPE IVEC = ARRAY [1 . . 3] OF COMPLEX;
CONST PIVEC = IVEC (PII, PII, COMPLEX (0.0, 1.0));
```

## 10.3 Numeric Constants

Numeric constants are irreducible numbers such as 45, 12.3, and 9E12. The notation of a numeric constant generally indicates its type: REAL, INTEGER, WORD, or INTEGER4.

Numbers can have a leading plus sign (+) or minus sign (-), except when the numbers are within expressions. Therefore:

```
ALPHA := +10   {Is legal}
ALPHA + -10    {Is illegal}
```

Blanks embedded within constants are not permitted.

The compiler truncates any number that exceeds a certain maximum number of characters and gives a warning when this occurs. The maximum length of constants (either 19 or 31) is the same as the maximum length of identifiers.

The syntax for numeric constants applies not only to the actual text of programs, but also to the content of textfiles read by a program.

Examples of numeric constants:

```
 123          0.17
+12.345       007
-1.7E-10     -26.0
 17E+3        26.0E12
-17E3         1E1
```

Numeric constants can appear in any of the following:

1. CONST sections
2. expressions
3. type clauses
4. set constants
5. structured constants
6. CASE statement CASE constants
7. variant record tag values

The different types of numeric constants are discussed in detail in the following sections.

## 10.3.1 REAL Constants

The type of a number is REAL if the number includes a decimal point or exponent. The REAL value range depends on the REAL number unit of the target machine. Generally, either the IEEE or the Microsoft REAL number format is used. This provides about seven digits of precision, with a maximum value of about 1.701411E38.

There is, however, a distinction between REAL values and REAL constants. The REAL constant range may be a subset of the REAL value range. In Microsoft format, REAL numeric constants must be greater than or equal to 1.0E–38 and less than 1.0E+38. In IEEE format, REAL numeric constants are kept in double precision and so can range from about 1E–306 to 1E+306.

The compiler issues a warning if there is not at least one digit on each side of a decimal point. A REAL number starting or ending with a decimal point may be misleading. For example, because left parenthesis-period substitutes for left square bracket, and right parenthesis-period for right square bracket, the following:

```
(.1 + 2.)
```

is interpreted as:

```
[1 + 2]
```

Scientific notation in REAL numbers (as in 1.23E-6 or 4E7) is supported. The decimal point and exponent sign are optional when an exponent is given. Both the uppercase "E" and the lowercase "e" are allowed in REAL numbers. "D" and "d" are also allowed to indicate an exponent. This provides compatibility with other languages.

When IEEE REAL4 and REAL8 format are used, all real constants are stored in REAL8 (double precision) format. If you require a single precision REAL4 constant, declare a REAL4 variable and give it your real constant value in a VALUE section. (You may wish to give this variable the READONLY attribute as well.)

Versions of the compiler that run on one machine but generate code for another may lose a small amount of significance in REAL constants.

## 10.3.2 INTEGER, WORD, and INTEGER4 Constants

The type of a non-REAL numeric constant is INTEGER, WORD, or INTEGER4. Table 10.1 shows the range of values that constants of each of these types can assume.

**Table 10.1**

**INTEGER, WORD, and INTEGER4 Constants**

| Type | Range of Values (minimum/maximum) | Predeclared Constant |
|---|---|---|
| INTEGER | -MAXINT to MAXINT | MAXINT=32767 |
| WORD | 0 to MAXWORD | MAXWORD=65535 |
| INTEGER4 | -MAXINT4 to MAXINT4 | MAXINT4=2147483647 |

MAXINT, MAXWORD, and MAXINT4 are all predeclared constant identifiers.

One of three things happens when you declare a numeric constant identifier:

1. A constant identifier from -MAXINT to MAXINT becomes an INTEGER.
2. A constant identifier from MAXINT+1 to MAXWORD becomes a WORD.
3. A constant identifier from -MAXINT4 to -MAXINT-1 or MAXWORD+1 to MAXINT4 becomes an INTEGER4.

However, any INTEGER type constant (including constant expressions and values from -32767 to -1) automatically changes to type WORD; if the INTEGER value is negative, 65536 is added to it and the underlying 16-bit value is not changed.

For example, you can declare a subrange of type WORD as WRD(0) . . 127; the upper bound of 127 is automatically given the type WORD. The reverse is not true; constants of type WORD are not automatically changed to type INTEGER.

The ORD and WRD functions also change the type of an ordinal constant to INTEGER or WORD. Also, any INTEGER or WORD constant automatically changes to type INTEGER4 if necessary, but the reverse is not true.

Examples of relevant conversions are given in Table 10.2.

**Table 10.2**

**Constant Conversions**

| Constant | Assumed Type |
|---:|---|
| 0 | INTEGER could become WORD or INTEGER4 |
| -32768 | INTEGER4 only |
| 32768 | WORD could become INTEGER4 |
| 0 . . 20000 | INTEGER subrange |
| 0 . . 50000 | WORD subrange |
| 0 . . 80000 | Invalid: no INTEGER4 subranges |
| -1 . . 50000 | Invalid: becomes 65535 . . 50000 (i.e., -1 is treated as 65536) |

At the standard level, any numeric constant (i.e., literal or identifier) may have a plus (+) or minus (-) sign.

### 10.3.3 Nondecimal Numbering

At the extend level, MS-Pascal supports not only decimal number notation, but also numbers in hexadecimal, octal, binary, or other base numbering (where the base can range from 2 to 36). The number sign (#) acts as a radix separator.

Examples of numbers in nondecimal notation:

```
16#FF02

10#987

 8#776

 2#111100
```

Leading zeros are recognized in the radix, so a number like 008#147 is permitted. In hexadecimal notation, upper or lowercase letters A through F are permitted. A nondecimal constant without the radix (such as #44) is assumed to be hexadecimal. Nondecimal notation does not imply a WORD constant and may be used for INTEGER, WORD, or INTEGER4 constants. You must not use nondecimal notation for REAL constants or numeric statement labels.

## 10.4 Character Strings

Most Pascal manuals refer to sequences of characters enclosed in single quotation marks as "strings." In MS-Pascal, they are called "string literals" to distinguish them from string constants, which may be expressions, or values of the STRING type.

A string constant contains from 1 to 255 characters. A string constant longer than one character is of type PACKED ARRAY [1 . . n] OF CHAR, also known in MS-Pascal as the type STRING (n). A string constant that contains just one character is of type CHAR. However, the type changes from CHAR to PACKED ARRAY [1 .. 1] OF CHAR (e.g., STRING (1)) if necessary. For example, a constant ('A') of type CHAR could be assigned to a variable of type STRING (1).

A literal apostrophe (single quotation mark) is represented by two adjacent single quotation marks (e.g., 'DON''T GO'). The null string (' ') is not permitted. A string literal must fit on a line. The compiler recognizes string literals enclosed in double quotation marks (" ") or accent marks (`), instead of single quotation marks, but issues a warning message when it encounters them.

The constant expression feature (discussed in Section 10.6, "Constant Expressions") permits string constants made up of concatenations of other string constants, including string constant identifiers, the CHR ( ) function, and structured constants of type STRING. This is useful for representing string constants that are longer than a line or that contain nonprinting characters. For example:

```
'THIS IS UNDERLINED' * CHR(13) * STRING (DO 18 OF '_')
```

The LSTRING feature of MS-Pascal adds the super array type LSTRING. LSTRING is similar to PACKED ARRAY [0 .. n] OF CHAR, except that element 0 contains the length of the string, which can vary from 0 to a maximum of 255. (See Section 7.2.2, "LSTRINGs," for a discussion of LSTRINGs.) For now, note that, if necessary, a constant of type STRING (n) or CHAR changes automatically to type LSTRING.

NULL is a predeclared constant for the null LSTRING, with the element 0 (the only element) equal to CHR (0). NULL cannot be concatenated, since it is not of type STRING. It is the only constant of type LSTRING.

Examples of string literal declarations:

```
NAME = 'John Jacob';          {a legal string literal}
LETTER = 'Z';                 {LETTER is of type
                              CHAR}
QUOTED_QUOTE = '''';          {quotes quote}
NULL_STRING = NULL;           {legal}
NULL_STRING = ' ';          {illegal}
DOUBLE = "OK";              {generates a warning}
```

## 10.5 Structured Constants

Standard Pascal permits only the numeric and string constants already mentioned, the pointer constant value NIL, and untyped constant sets.

At the extend level of MS-Pascal, however, you may use constant arrays, records, and typed sets. Structured constants can be used anywhere a structured value is allowed, in expressions as well as in CONST and VALUE sections.

1. An array constant consists of a type identifier followed by a list of constant values in parentheses separated by commas.

   Example of an array constant:

   ```
   TYPE VECT_TYPE = ARRAY [-2 . . 2] OF INTEGER;
   CONST VECT = VECT_TYPE (5, 4, 3, 2, 1);
   VAR A : VECT_TYPE;
   VALUE A := VECT;
   ```

2. A record constant consists of a type identifier followed by a list of constant values in parentheses separated by commas.

   Example of a record constant:

   ```
   TYPE REC_TYPE = RECORD
               A, B : BYTE;
               C, D : CHAR
           END;
   CONST RECR = REC_TYPE (#20, 0, 'A', CHR (20));
   VAR FOO : REC_TYPE;
   VALUE FOO := RECR;
   ```

3. A set constant consists of an optional set type identifier followed by set constant elements in square brackets. Set constant elements are separated by commas. A set constant element is either an ordinal constant, or two ordinal constants separated by two dots to indicate a range of constant values.

Example of a set constant:

```
TYPE COLOR_TYPE = SET OF
    (RED, BLUE, WHITE, GREY, GOLD);
CONST SETC = COLOR_TYPE [RED, WHITE .. GOLD];
VAR RAINBOW : COLOR_TYPE;
VALUE RAINBOW := SETC;
```

A constant within a structured array or record constant must have a type that can be assigned to the corresponding component type. For records with variants, the value of a constant element corresponding to a tag field selects a variant, even if the tag field is empty. The number of constant elements must equal the number of components in the structure, except for super array type structured constants. Nested structured constants are permitted.

An array or record constant nested within another structured constant must still have the preceding type identifier. For this reason, a super array constant can have only one dimension (see Section 7.2, "Super Arrays," for a discussion of super arrays). The size of the representation of a structured constant must be from 1 to 255 bytes. If this 255-byte limit is a problem, declare a structured variable with the READONLY attribute, and initialize its components in a VALUE section.

Example of a complex structured constant:

```
TYPE R3 = ARRAY [1 .. 3] OF REAL;
TYPE SAMPLE = RECORD  I : INTEGER;
                      A : R3;
                      CASE BOOLEAN OF
                      TRUE : (S : SET OF 'A' .. 'Z';
                              P : ^SAMPLE);
                      FALSE : (X : INTEGER)
                 END;
CONST SAMP_CONST = SAMPLE (27, R3 (1.4, 1.4, 1.4),
                 TRUE, ['A','E','I'], NIL);
```

Constant elements can be repeated with the phrase DO *n* OF *constant*, so the previous example could have included "DO 3 OF 1.4" instead of "1.4, 1.4, 1.4".

MS-Pascal does not support set constant expressions, such as ['_'] + LETTERS, or file constant expressions. The constant

'ABC' of type STRING (3) is equivalent to the structured constant STRING ('A', 'B', 'C'). LSTRING structured constants are not permitted; use the corresponding STRING constants instead.

Structured constants (and other structured values, such as variables and values returned from functions) can be passed by reference using CONST parameters. For more information, see Section 14.4, "Procedure and Function Parameters."

There are two kinds of set constants: one with an explicit type, as in CHARSET ['A' . . 'Z'], and one with an unknown type, as in [20 . . 40]. You may use either in an expression or to define the value of a constant identifier. Set constants with an explicit type may also be passed as a reference (CONST) parameter. Sets of unknown type are unpacked, but the type changes to PACKED if necessary. Passing sets by reference is generally more efficient than passing them as value parameters.

## 10.6 Constant Expressions

Constant expressions in MS-Pascal allow you to compute constants based on the values of previously declared constants in expressions. Constant expressions can also occur within program statements.

Example of a constant expression declaration:

```
CONST HEIGHT_OF_LADDER = 6;
      HEIGHT_OF_MAN = 6;
      REACH = HEIGHT_OF_LADDER + HEIGHT_OF_MAN;
```

Because a constant expression may contain only constants that you have declared earlier, the following is illegal:

```
CONST MAX = A + B;
      A = 10;
      B = 20;
```

Certain functions may be used within constant expressions. For example:

```
CONST A = LOBYTE (-23) DIV 23;
      B = HIBYTE (-A);
```

Table 10.3 shows the functions and operators you may use with REAL, INTEGER, WORD, and other ordinal constants, such as enumerated and subrange constants.

**Table 10.3**

**Constant Operators and Functions**

| Type of Operand | Functions and Operators | | | |
|---|---|---|---|---|
| REAL, INTEGER | Unary plus (+)<br>Unary minus (-) | | | |
| INTEGER, WORD | +<br>-<br>* | DIV<br>MOD<br>AND | OR<br>NOT<br>XOR | HIBYTE( )<br>LOBYTE( )<br>BYWORD( ) |
| Ordinal types | <<br>><br>= | <=<br>>=<br><> | CHR( )<br>ORD( )<br>WRD( ) | LOWER( )<br>UPPER( ) |
| Boolean | AND | | NOT | OR |
| ARRAY | LOWER( ) | | UPPER( ) | |
| Any type | SIZEOF( ) | | RETYPE( ) | |

Examples of constant expressions:

```
CONST FOO = (100 + ORD('X')) * 8#100 + ORD('Y');
      MAXSIZE = 80;
      X = (MAXSIZE > 80) OR (IN_TYPE = PAPERTAPE);
      {X is a BOOLEAN constant}
```

In addition to the operators shown in Table 10.3 for numeric constants, you may use the string concatenation operator (*) with string constants, as follows:

```
CONST A = 'abcdef';
      M = CHR (109); {CHR is allowed}
      ATOM = A * 'ghijkl' * M;
      {ATOM = 'abcdefghijklm'}
```

These constants can span more than one line, but are still limited to the 255 character maximum. These string constant expressions are allowed wherever a string literal is allowed, except in metacommands.

# Chapter 11
# Variables and Values

This chapter describes Microsoft Pascal variable types and the ways you can set their values and attributes.

## 11.1 What Is a Variable?

A variable is a value that is expected to change during the course of a program. Every variable must be of a specific data type. A variable may have an identifier.

If A is a variable of type INTEGER, then the use of A in a program actually refers to the data denoted by A. For example:

```
VAR A : INTEGER;
    BEGIN
        A := 1;
        A := A + 1
    END;
```

These statements would first assign a value of 1 to the data denoted by A, and subsequently assign it a value of 2.

Variables are manipulated by using some sort of notation to denote the variable, in the simplest case, a variable identifier. In other cases, variables may be denoted by array indices or record fields or the dereferencing of pointer or address variables.

The compiler itself may sometimes create "hidden" variables, allocated on the stack, in circumstances like the following:

1. When you call a function that will return a structured result, the compiler allocates a variable in the caller for the result.
2. When you need the address of an expression (e.g., to pass it as a reference parameter or to use it as a WITH statement record or with ADR or ADS), the compiler allocates a variable for the value of the expression.
3. The initial and final values of a FOR loop may require allocating a variable.
4. When the compiler evaluates an expression, it may allocate a variable to store intermediate results.

5. Every WITH statement requires a variable to be allocated for the address of the WITH's record.

## 11.2 Declaring a Variable

A variable declaration consists of the identifier for the new variable, followed by a colon and a type. You may declare variables of the same type by giving a list of the variable identifiers, followed by their common type. For example:

```
VAR XCOORD, YCOORD : REAL
```

You may declare a variable in any of the following locations:

1. the VAR section of a program, procedure, function, module, interface, or implementation
2. the formal parameter list of a procedure, function, or procedural parameter

In a VAR section, you may declare a variable to be of any legal type; in a formal parameter list, you may include only a type identifier (i.e., you may not declare a type in the heading of a procedure or function). For example:

```
PROCEDURE NAME (GEORGE : ARRAY [1 . . 10] OF COLOR)
{Illegal; GEORGE is of a new type.}

VAR VECTOR_A : VECTOR (10)
{Legal; VECTOR (10) is a type derived from}
{a super type.}
```

Each declaration of a file variable F of type FILE OF T implies the declaration of a buffer variable of type T, denoted by F^. At the extend level, a file declaration also implies the declaration of a record variable of type FCBFQQ, whose fields are denoted as F.TRAP, F.ERRS, F.MODE, and so on. See Section 8.2, "The Buffer Variable," and Section 8.6, "File I/O: Extend Level," for further information on buffer variables and FCBFQQ fields, respectively.

## 11.3 The VALUE Section

The VALUE section in Microsoft Pascal lets you give initial values to variables in a program, module, procedure, or function. You may also initialize the variable in an implementation, but not in an interface.

The VALUE section may include only statically allocated variables, that is, any variable declared at the program, module, or implementation level, or a variable with the STATIC or PUBLIC attribute. Variables with the EXTERN or ORIGIN attribute cannot occur in a VALUE section, since they are not allocated by the compiler.

The VALUE section may contain assignments of constants to entire variables or to components of variables. For example:

```
VAR ALPHA : REAL;
    ID : STRING (7);
    I : INTEGER;

VALUE ALPHA := 2.23;
    ID[1] := 'J';
    I := 1;
```

However, within a VALUE section, you may not assign a variable to another variable. The last line in the following example is illegal, since "I" must be a constant:

```
CONST MAX = 10;
VAR I, J : INTEGER;
VALUE I  := MAX;
      J := I;
```

If the $rom metacommand is off, variables are initialized by loading the static data segment. If the $rom metacommand is on, the VALUE section generates an error message since ROM-based systems usually cannot statically initialize data.

## 11.4 Using Variables and Values

At the standard level of MS-Pascal, denotation of a variable may designate one of three things:

1. an entire variable
2. a component of a variable
3. a variable referenced by a pointer

A value may be any of the following:

1. a variable
2. a constant
3. a function designator
4. a component of a value
5. a variable referenced by a reference value

At the extend level, a function can also return an array, record, or set. The same syntax used for variables may be used to denote components of the structures these functions return.

This feature also allows you to dereference a reference type that is returned by a function. However, you may only use the function designator as a value, not as a variable. For example, the following is illegal:

```
    F (X, Y)^ := 42;
```

Also at the extend level, you may declare constants of a structured type. Components of a structured constant use the same syntax as variables of the same type. See Section 10.6, "Constant Expressions," for further discussion of this topic.

Examples of structured constant components:

```
TYPE REAL3 = ARRAY [1 .. 3] OF REAL;
{an array type}
CONST PIES = REAL3 (3.14, 6.28, 9.42);
{an array constant}
   .
   .
   X := PIES [1] * PIES [3];
   {i.e., 3.14 * 9.42}
   Y := REAL3 (1.1, 2.2, 3.3) [2];
   {i.e., 2.2}
```

## 11.4.1 Components of Entire Variables and Values

At the standard level, a variable identifier denotes an entire variable. A variable, function designator, or constant denotes an entire value.

A component of a variable or value is denoted by the identifier followed by a selector that specifies the component. The form of a selector depends on the type of structure (array, record, file, or reference).

### 11.4.1.1 Indexed Variables and Values

A component of an array is denoted by the array variable or value, followed by an index expression. The index expression must be assignment compatible with the index type in the array type declaration. An index type must always be an ordinal type. The index itself must be enclosed in brackets following the array identifier.

Examples of indexed variables and values:

```
ARRAY_OF_CHAR ['C']
{Denotes the Cth element.}

'STRING CONSTANT' [6]
{Denotes the 6th element, the letter 'G'.}

BETAMAX [12] [-3]
BETAMAX [12,-3]
{These two say the same thing.}
```

```
ARRAY_FUNCTION (A, B) [C, D]
{Denotes a component of a two-dimensional array}
{returned by ARRAY_FUNCTION (A, B). A and B are}
{actual parameters.}
```

You may specify the current length of an LSTRING variable, LSTR, in either of two ways:

1. with the notation LSTR [0], to access the length as a CHAR component
2. with the notation LSTR.LEN, to access the length as a BYTE value

### 11.4.1.2 Field Variables and Values

A component of a record is denoted by the record variable or value followed by the field identifier for the component. Fields are separated by the period (.). In a WITH statement, you give the record variable or value only once. Within the WITH statement, you may use the field identifier of a record variable directly.

Examples of field variables and values:

```
PERSON.NAME := 'PETE'

PEOPLE.DRIVERS.NAME := 'JOAN'

WITH PEOPLE.DRIVERS DO NAME := 'GERI'

RECURSING_FUNC ('XYZ').BETA
{Selects BETA field of record returned}
{by the function named RECURSIVE_FUNC.}

COMPLEX_TYPE (1.2, 3.14).REAL_PART
```

Record field notation also applies to files for FCBFQQ fields, to address type values for numeric representations, and to LSTRINGs for the current length.

### 11.4.1.3 File Buffers and Fields

At any time, only one component of a file is accessible. The accessible component is determined by the current file position and represented by the buffer variable. Depending on the status of the buffer variable, the fetching process may first read the value from the file. (This is called "lazy evaluation"; see Section 16.1.5, "Lazy Evaluation," for details.)

If a file buffer variable is passed as a reference parameter or used as a record of a WITH statement, the compiler issues a warning to alert you to the fact that the value of the buffer variable may not be correct after the position of the file is changed with a GET or PUT procedure.

Examples of file reference variables:

```
INPUT^
ACCOUNTS_PAYABLE.FILE^
```

## 11.4.2 Reference Variables

Reference variables or values denote data that refers to some data type. There are three kinds of reference variables and values:

1. pointer variables and values
2. ADR variables and values
3. ADS variables and values

In general, a reference variable or value "points" to a data object. Thus, the value of a reference variable or value is a reference to that data object. To obtain the actual data object pointed to, you must "dereference" the reference variable by appending an up arrow (^) to the variable or value.

Example using pointer values:

```
VAR P, Q : ^INTEGER;
{P and Q are pointers to integers.}

NEW (P); NEW (Q);
{P and Q are assigned reference values to}
{regions in memory corresponding to data}
{objects of type INTEGER.}

P := Q;
{P and Q now point to the same region}
{in memory.}
```

```
P^ := 123;
{Assigns the value 123 to the INTEGER value}
{pointed to by P. Since Q points to this}
{location as well, Q^ is also assigned 123.}
```

Using NIL^ is an error (since a NIL pointer does not reference anything). At the extend level, you may also append an up arrow (^) to a function designator for a function that returns a pointer or address type. In this case, the up arrow denotes the value referenced by the return value. This variable cannot be assigned to or passed as a reference parameter.

Examples of functions returning reference values:

```
DATA1 := FUNK1 (I, J)^
{FUNK1 returns a reference value. The up arrow}
{dereferences the reference value returned,}
{assigning the referenced data to DATA1.}

DATA2 := FUNK2 (K, L)^.FOO [2]
{FUNK2 returns a reference value. The up arrow}
{dereferences the reference value returned. In}
{this case, the dereferenced value is a record.}
{The array component FOO [2] of that record is}
{assigned to the variable DATA2.}
```

If P is of type ADR OF some type, then P.R denotes the address value of type WORD. If P is of type ADS OF some type, then P.R denotes the offset portion of the address and P.S denotes the segment portion of the address. Both portions are of type WORD.

Examples of address variables:

```
BUFF_ADR.R
DATA_AREA.S
```

## 11.5 Attributes

At the extend level of MS-Pascal, a variable declaration or the heading of a procedure or function may include one or more attributes. A variable attribute gives special information about the variable to the compiler.

Table 11.1 displays the attributes provided by MS-Pascal for variables.

**Table 11.1**

**Attributes for Variables**

| Attribute | Variable |
|---|---|
| STATIC | Allocated at a fixed location, not on the stack |
| PUBLIC | Accessible by other modules with EXTERN, implies STATIC |
| EXTERN | Declared PUBLIC in another module, implies STATIC |
| ORIGIN | Located at specified address, implies STATIC |
| PORT | I/O address, implies STATIC |
| READONLY | Cannot be altered or written to |

The EXTERN attribute is also a procedure and function directive; PUBLIC and ORIGIN are also procedure and function attributes. See Section 14.3, "Attributes and Directives," for a discussion of procedure and function attributes and directives. The following sections, 11.5.1 through 11.5.5, discuss each of the variable attributes in detail.

You may only give attributes for variables in a VAR section. Specifying variable attributes in a TYPE section or a procedure or function parameter list is not permitted.

You give one or more attributes in the variable declaration, enclosed in brackets and separated by commas (if specifying more than one attribute).

The brackets may occur in either of two places:

1. An attribute in brackets after a variable identifier in a VAR section applies to that variable only.
2. An attribute in brackets after the reserved word VAR applies to all of the variables in the section.

Examples that specify variable attributes:

```
VAR A, B, C [EXTERN] : INTEGER;
{Applies to C only.}

VAR [PUBLIC] A, B, C : INTEGER;
{Applies to A, B, and C.}

VAR [PUBLIC] A, B, C [ORIGIN 16#1000] : INTEGER;
{A, B, and C are all PUBLIC. ORIGIN of C}
{is the absolute hexadecimal address 1000.}
```

## 11.5.1 The STATIC Attribute

The STATIC attribute gives a variable a unique, fixed location in memory. This is in contrast to a procedure or function variable that is allocated on the stack or one that is dynamically allocated on the heap. Use of STATIC variables can save time and code space, but increases data space.

All variables at the program, module, or unit level are automatically assigned a fixed memory location and given the STATIC attribute.

Functions and procedures that use STATIC variables can execute recursively, but STATIC variables must be used only for data common to all invocations. Since most of the other variable attributes imply the STATIC attribute, the trade-off between savings in time and code space or reduced data space applies to the PUBLIC, EXTERN, ORIGIN, and PORT attributes as well.

Files declared in a procedure or function with the STATIC attribute are initialized when the routine is entered; they are closed when the routine terminates like other files. However, other STATIC variables are only initialized before program execution. This means that, except for open FILE variables, STATIC variables can be used to retain values between invocations of a procedure or function.

Examples of STATIC variable declarations:

```
VAR VECTOR [STATIC] : ARRAY [0 . . MAXVEC] OF INTEGER;
VAR [STATIC] I, J, K : 0 . . MAXVEC;
```

The STATIC attribute does not apply to procedures or functions, as do some other attributes.

## 11.5.2 The PUBLIC and EXTERN Attributes

The PUBLIC attribute indicates a variable that may be accessed by other loaded modules; the EXTERN attribute identifies a variable that resides in some other loaded module. The identifier is passed to the target linker in the generated code object file (where it may be truncated if the linker imposes a length restriction).

Variables given the PUBLIC or EXTERN attribute are implicitly STATIC.

Examples of PUBLIC and EXTERN variable declarations:

```
VAR [EXTERN] GLOBE1, GLOBE2 : INTEGER;
{The variables GLOBE1 and GLOBE2 are declared}
{EXTERN, meaning that they must be declared}
{PUBLIC in some other loaded module.}

VAR BASE_PAGE [PUBLIC, ORIGIN #12FE] : BYTE;
{The variable BASE_PAGE is located at 12FE,}
{hexadecimal. Because it is also PUBLIC, it can}
{be accessed from other loaded modules that}
{declare BASE_PAGE with the EXTERN attribute.}
```

PUBLIC variables are usually allocated by the compiler, unless you also give them an ORIGIN. Giving a variable both the PUBLIC and ORIGIN attributes tells the loader that a global name has an absolute address. PUBLIC cannot be combined with PORT.

If both PUBLIC and ORIGIN are present, the compiler does not need the loader to resolve the address. However, the identifier is still passed to the linker for use by other modules.

EXTERN variables are not allocated by the compiler. Nor do they have an ORIGIN, since giving both EXTERN and ORIGIN implies two different ways to access the variable. The reserved word EXTERNAL is synonymous with EXTERN. This increases portability from other Pascals, since others commonly use one of the two.

Variables in the interface of a unit are automatically given either the PUBLIC or EXTERN attribute. If a program, module, or unit USES an interface, its variables are made EXTERN; if you compile the IMPLEMENTATION of the interface, its variables are made PUBLIC.

## 11.5.3 The ORIGIN and PORT Attributes

The ORIGIN attribute directs the compiler to locate a variable at a given memory address; the PORT attribute specifies some kind of I/O address. In either case, the address must be a constant of any ordinal type. I/O ports, interrupt vectors, operating system data, and other related data can be accessed with ORIGIN or PORT variables.

Examples of ORIGIN and STATIC variable declarations:

```
VAR KEYBOARDP [PORT 16#FFF2] : CHAR;
VAR INTRVECT [ORIGIN 8#200] : WORD;
```

Variables with ORIGIN or PORT attributes are implicitly STATIC. Also, they inhibit common subexpression optimization. For example, if GATE has the ORIGIN attribute, the two statements X := GATE; Y := GATE; access GATE twice in the order given, instead of using the first value for both assignments. This ensures correct operation if GATE is a memory-mapped input port. However, if GATE is passed as a reference parameter, references to the parameter may be optimized away. For this reason, PORT variables cannot be passed as reference parameters.

ORIGIN and PORT variables are never allocated or initialized by the compiler. The associated address only indicates where the variable is found. ORIGIN always implies a memory address, but the meaning of PORT varies with the implementation.

In most implementations, I/O is assumed to be memory mapped, so PORT is just a synonym for ORIGIN. Other implementations use the machine's native input and output instructions. Still others call port input and output routines for every access.

Giving the PORT and ORIGIN attributes in brackets immediately following the VAR keyword is ambiguous and generates an error during compilation. (It would be unclear to the compiler whether all variables following should be at the same address or whether addresses should be assigned sequentially.)

```
VAR [ORIGIN 0] FIRST, SECOND : BYTE; {ILLEGAL!}
```

ORIGIN (but not PORT) permits a segmented address using "segment: offset" notation.

```
VAR SEGVECT [ORIGIN 16#0001 : 16#FFFE] : WORD;
```

Currently, a variable with a segmented ORIGIN cannot be used as the control variable in a FOR statement.

## 11.5.4 The READONLY Attribute

The READONLY attribute prevents assignments to a variable. It also prevents the variable being passed as a VAR or VARS parameter. Also, a READONLY variable cannot be read with a READ statement or used as a FOR control variable. You may use READONLY with any of the other attributes.

Examples of READONLY variable declarations:

```
VAR INPORT [PORT 12, READONLY] : BYTE;
{INPORT is a READONLY PORT variable.}

VAR [READONLY] I, J [PUBLIC], K [EXTERN] : INTEGER;
{I, J, and K are all READONLY;}
{J is also PUBLIC; K is also EXTERN.}
```

CONST and CONSTS parameters, as well as FOR loop control variables (while in the body of the loop), are automatically given the READONLY attribute. READONLY is the only variable attribute that does not imply STATIC allocation. A variable that is both READONLY and either PUBLIC or EXTERN in one source file is not necessarily READONLY when used in another source file. The READONLY attribute does not apply to procedures and functions.

## 11.5.5 Combining Attributes

You may give a variable multiple attributes. Separate the attributes with commas and enclose the list in brackets, as shown:

```
VAR [STATIC]
X, Y, Z [ORIGIN #FFFE, READONLY] : INTEGER;
```

In this example, Z is a STATIC, READONLY variable with an ORIGIN at hexadecimal FFFE. These rules apply when you are combining attributes:

1. If you give a variable the EXTERN attribute, you may not give it the PORT, ORIGIN, or PUBLIC attributes in the current compiland.
2. If you give a variable the PORT attribute, you may not give it the ORIGIN, PUBLIC, or EXTERN attributes at all.
3. If you give a variable the ORIGIN attribute, you may not give it the PORT or EXTERN attributes. However, you may combine ORIGIN with PUBLIC.
4. If you give a variable the PUBLIC attribute, you may not give it the PORT or EXTERN attributes. However, you may combine PUBLIC with ORIGIN.
5. You may use STATIC and READONLY with any other attributes.

# Chapter 12
# Expressions

Expressions are constructions that evaluate to values. Table 12.1 illustrates a variety of expressions, which, if A = 1 and B = 2, evaluate to the value shown.

**Table 12.1**

**Expressions**

| Expression | Value |
|---|---|
| 2 | 2 |
| A | 1 |
| A + 2 | 3 |
| (A + 2) | 3 |
| (A + 2) * (B - 3) | -3 |

The operands in an expression may be a value or any other expression. When any operator is applied to an expression, that expression is called an operand. With parentheses for grouping and operators that use other expressions, you can construct expressions as long and complicated as desired.

The available operators, in the order in which they are applied, are as follows:

1. Unary

   NOT [ADR ADS]

2. Multiplying

   * / DIV MOD AND (ISR SHL SHR)

3. Adding

   + - OR (XOR)

4. Relational

   = <> <= >= < > IN

Operators shown above in parentheses are available only at the extend level of Microsoft Pascal, those in brackets only at the system level.

A standard Pascal expression is either a value or the result of applying an operator to one or two values. Although a value can be of almost any type, most MS-Pascal operators only apply to the following types:

| | |
|---|---|
| INTEGER | INTEGER4 |
| WORD | BOOLEAN |
| REAL | SET |

The relational operators also apply for the CHAR, enumerated, string, and reference types. For all operators (except the set operator IN), operands must have compatible types.

## 12.1 Simple Type Expressions

As a rule, the operands and the value resulting from an operation are all of the same type. Occasionally, however, the type of an operand is changed to the type required by an operator.

This conversion occurs on two levels: one for constant operands only, and one for all operands. INTEGER to WORD conversion occurs for constant operands only; conversion from INTEGER to REAL and from INTEGER or WORD to INTEGER4 occurs for all operands.

If necessary in constant expressions, INTEGER values change to WORD type. Be careful when mixing INTEGER and WORD constants in expressions. For example, if CBASE is the constant 16#C000 and DELTA is the constant -1, the following expression gives a WORD overflow:

```
WRD (CBASE) + DELTA
```

The overflow occurs because DELTA is converted to the WORD value 16#FFFF, and 16#C000 plus 16#FFFF is greater than MAXWORD. However, the following would work:

```
WRD (ORD (CBASE) + DELTA)
```

This expression gives the INTEGER value -16385, which changes to WORD 16#BFFF. If conversion is needed by an operator or for an assignment, the compiler makes the following conversions:

1. from INTEGER to REAL or INTEGER4
2. from WORD to INTEGER4

The following rules determine the type of the result of an expression involving these simple types:

1. + - *

   These operators affect INTEGERs, REALs, WORDs, and INTEGER4s, as shown in the following examples:

   +123

   A + 123

   -23.4

   A - 8

   A * B * 3

   Mixtures of REALs with INTEGERs and of INTEGER4s with INTEGERs or WORDs are allowed. Where both operands are of the same type, the resulting type is the type of the operands. If either operand is REAL, the resulting type is REAL; otherwise, if either operand is INTEGER4, the resulting type is INTEGER4.

   Unary plus (+) and minus (-) are supported, along with the binary forms. Unary minus on a WORD type is two's complement (NOT is one's complement); since there are no negative WORD values, this always generates a warning.

   Because unary minus has the same precedence level as the adding operators:

   ```
   (X * -1) {Is legal}
   (-256 AND X) {Is interpreted as -(256 AND X)}
   ```

2. /

   This is a "true" division operator. The result is always REAL. Operands may be INTEGER or REAL (not WORD or INTEGER4).

Examples of division:

```
34 / 26.4 = 1.28787 . . .
18 / 6    = 3.00000 . . .
```

3. DIV MOD

These are the operators for integer divide quotient and remainder, respectively. The left operand (dividend) is divided by the right operand (divisor).

Examples of integer division:

```
 123 MOD  5 =  3
-123 MOD  5 = -3 {Sign of result is}
                 {sign of dividend}
 123 MOD -5 =  3
 1.3  MOD  5      {Illegal with REAL operands}
 123 DIV   5 = 24
 1.3  DIV   5      {Illegal with REAL operands}
```

Both operands must be of the same type : INTEGER, WORD, or INTEGER4 (not REAL). The sign of the remainder (MOD) is always the sign of the dividend.

MS-Pascal differs from the current draft ISO standard with respect to the semantics for DIV and MOD with negative operands, but the resulting code is more efficient. Programs intended to be portable should not use DIV and MOD unless both operands are positive.

4. AND OR XOR NOT

These extend level operators are bitwise logical functions. Operands must be INTEGER or WORD or INTEGER4 (never a mixture), and cannot be REAL. The result has the type of the operands.

NOT is a bitwise one's complement operation on the single operand. If an INTEGER variable V has the value MAXINT, NOT V gives the illegal INTEGER value -32768. This generates an error if the initialization switch is on and the value is used later in a program.

Given the following initial INTEGER values,

X = 2#1111000011110000
Y = 2#1111111100000000

AND, OR, XOR, and NOT perform the following functions:

```
X AND Y     1111000011110000
            1111111100000000
            ----------------
            1111000000000000

X OR Y      1111000011110000
            1111111100000000
            ----------------
            1111111111110000

X XOR Y     1111000011110000
            1111111100000000
            ----------------
            0000111111110000

NOT X       1111000011110000
            ----------------
            0000111100001111
```

5. SHL SHR ISR

These extend level operators provide bitwise shifting functions.

SHL and SHR are logical shifts left and right. ISR is an integer (signed) arithmetic shift right: the sign bit is always propagated, even on a WORD type operand. Since the compiler cannot generate a simple right shift for INTEGER division (-1 DIV 2 would be incorrect) and division is a very time-consuming operation, SHR or ISR could be used instead of DIV where appropriate.

Operands must be both INTEGER, both WORD, or both INTEGER4; they cannot be REAL. The result has the same type as the operands.

The left operand is shifted, and the right operand is the shift count in bits. A shift count less than 0 or greater than 32 produces undefined results and generates an error message if the range checking switch is on. Shifts never cause overflow errors; shifted bits are simply lost.

Given that X = 2#1111111100000000, the shifting functions would perform the following operations:

```
X          1111111100000000
X SHL 1    1111111000000000
X SHR 1    0111111110000000
X ISR  1   1111111110000000 {sign extension}
```

## 12.2 Boolean Expressions

The Boolean operators at the standard level of MS-Pascal are:

| | | |
|---|---|---|
| NOT | AND | OR |
| = | < | > |
| < > | <= | >= |

XOR is available at the extend and systems levels.

You may also use P <> Q as an exclusive OR function. Since FALSE < TRUE, P <= Q denotes the Boolean operation "P implies Q." Furthermore, the Boolean operators AND and OR are not the same as the WORD and INTEGER operators of the same name that are bitwise logical functions. The Boolean AND and OR operators may or may not evaluate their operations. The following example illustrates the danger of assuming that they don't:

```
WHILE (I <= MAX) AND (V [I] <> T) DO I := I + 1;
```

If array V has an upper bound MAX, then the evaluation of V [I] for I > MAX is a runtime error. This evaluation may or may not take place. Sometimes both operands are evaluated during optimization, and sometimes the evaluation of one may cause the evaluation of the other to be skipped. In the latter case, either operand may be evaluated first.

Instead, use the following construction:

```
WHILE I <= MAX DO
    IF V [I] <> T THEN I := I + 1 ELSE BREAK;
```

See Section 13.3.5, "Sequential Control," for information on using AND THEN and OR ELSE to handle situations, such as the previous example, where tests are examined sequentially.

The relational operators produce a Boolean result. The types of the operands of a relational operator (except for IN) must be compatible. If they are not compatible, one must be REAL and the other compatible with INTEGER.

Reference types can only be compared with = and <>. To compare an address type with one of the other relational operators, you must use address field notation, as shown:

```
IF (A.R < B.R) THEN statement;
```

Except for the string types STRING and LSTRING, you cannot compare files, arrays, and records as wholes. Two STRING types must have the same upper bound to be compared; two LSTRINGs may have different upper bounds.

In LSTRING comparison, characters past the current length are ignored. If the current length of one LSTRING is less than the length of the other and all characters up to the length of the shorter are equal, the compiler assumes the shorter one is "less than" the longer one. However, two LSTRINGs are not considered equal unless all current characters are equal and their current lengths are equal.

The six relational operators =, <>, <=, >=, <, and > have their normal meaning when applied to numeric, enumerated, CHAR, or string operands. Section 12.3, "Set Expressions," discusses the meaning of these relational operators (along with the relational operator IN) when applied to sets. Since the relational operators in Boolean expressions have a lower precedence than AND and OR, the following is incorrect:

```
IF I < 10 AND J = K THEN
```

Instead, you must write:

```
IF (I < 10) AND (J = K) THEN
```

Also, you may not use the numeric types where a Boolean operand is called for. (Some other languages permit this.) For an integer I, the clause IF I THEN is illegal; you must use the following instead:

```
IF I <> 0 THEN
```

Note, however, that MS-Pascal does allow the following:

```
$if I $then
```

The inclusion of special NAN ("Not A Number") values means that a comparison between two real numbers can have a result other than less-than, equal, or greater-than. The numbers can be unordered, meaning one or both are NANs. An unordered result is the same as "not equal, not less than, and not greater than."

For example, if variables A or B are NAN values:

1. A < B is false.
2. A <= B is false.
3. A > B is false.
4. A >= B is false.
5. A = B is false.
6. A <> B is, however, true.

REAL comparisons do not follow the same rules as other comparisons in many ways. A < B is not always the same as NOT (B <= A); this prevents some optimizations otherwise done by the compiler. If A is a NAN, then A <> A is true; in fact, this is a good way to check for a NAN value.

## 12.3 Set Expressions

Table 12.2 shows the MS-Pascal operators that apply differently to sets than to other types of expressions.

**Table 12.2**

**Set Operators**

| Operator | Meaning in Set Operations |
|---|---|
| + | Set union |
| - | Set difference |
| * | Set intersection |
| = | Test set equality |
| <> | Test set inequality |
| <= and >= | Test subset and superset |
| < and > | Test proper subset and superset |
| IN | Test set membership |

Any operand whose type is SET OF S, where S is a subrange of T, is treated as if it were SET OF T. (T is restricted to the range from 0 to 255 or the equivalent ORD values.) Either both operands must be PACKED or neither must be PACKED, unless one operand is a constant or constructed set.

With the IN operator, the left operand (an ordinal) must be compatible with the base type of the right operand (a set). The expression X IN B is TRUE if X is a member of the set B, and FALSE otherwise. X can be outside of the range of the base type of B legally. For example, X IN B is always false if the following statements are true:

```
X = 1
B = SET OF 2 . . 9
```

(1 is compatible, but not assignment compatible, with 2 . . 9).

Angle brackets are set operators only at the extend level of MS-Pascal, since the ISO standard does not support them for sets. They test that a set is a proper subset or superset of another set. Proper subsetting does not permit a set as a subset if the two sets are equal.

Expressions involving sets may use the "set constructor," which gives the elements in a set enclosed in square brackets. Each element can be an expression whose type is in the base type of the set or the lower and upper bounds of a range of elements in the base type. Elements cannot be sets themselves.

Examples of sets involving set constructors:

```
SET_COLOR := [RED, BLUE . . PURPLE] - [YELLOW]

SET_NUMBER :=
     [12, J+K, TRUNC (EXP (X)) . . TRUNC (EXP (X+1))]
```

Set constructor syntax is similar to CASE constant syntax. If X > Y then [X . . Y] denotes the empty set. Empty brackets also denote the empty set and are compatible with all sets. Also, if all elements are constant, a set constructor is the same as a set constant.

Like other structured constants, the type identifier for a constant set can be included in a set constant, as in COLORSET [RED . . BLUE]. This does not mean that a set constructor with variable elements can be given a type in an expression: NUMBERSET [I . . J] is illegal if I or J is a variable.

A set constructor such as [I, J, . . K] or an untyped set such as [1, 5 . . 7], is compatible with either a PACKED or an unpacked set. A typed set constant, such as DIGITS [1, 5 . . 7], is only compatible with sets that are PACKED or unpacked, respectively, in the same way as the explicit type of the constant.

## 12.4 Function Designators

A function designator specifies the activation of a function. It consists of the function identifier, followed by a (possibly empty) list of "actual parameters" in parentheses:

```
{Declaration of the function ADD.}
FUNCTION ADD (A, B : INTEGER); INTEGER;
    .
    .
{Use of the function ADD in an expression.}
X : = ADD (7, X * 4) + 123;
{ADD is a function designator.}
```

These actual parameters substitute, position for position, for their corresponding "formal parameters," defined in the function declaration.

Parameters can be variables, expressions, procedures, or functions. If the parameter list is empty, the parentheses must be omitted. See Section 14.4, "Procedure and Function Parameters," for more information on parameters.

The order of evaluation and binding of the actual parameters varies, depending on the optimizations used. If the $simple metacommand is on, the order is left to right.

In most computer languages, functions have two different uses:

1. In the mathematical sense, they take one or more values from a domain to produce a resulting value in a range. In this case, if the function never does anything else (such as assign to a global variable or do input/output), it is called a "pure" function.
2. The second type of function may have side effects, such as changing a static variable or a file. Functions of this second kind are said to be "impure."

At the standard level, a function may return either a simple type or a pointer. At the extend level, a function can return any assignable type (i.e., any type except a file or super array).

At the standard level, a pointer returned by a function can only be compared, as signed, or passed as a value parameter. At the extend level, however, the usual selection syntax for reference types, arrays, and records is allowed, following the function designator. See Section 11.4, "Using Variables and Values," for more information.

Examples of function designators:

```
SIN (X + Y)

NEXTCHAR

NEXTREC (17)^
{Here the function return type}
{is a pointer, and the returned}
{pointer value is dereferenced.}
```

```
NAD.NAME [1]
{Here the function has no parameters.}
{The return type is a record, one}
{field of which is an array.}
{The identifier for that field is}
{NAME. The example above selects}
{the first array component of the}
{returned record.}
```

It is more efficient to return a component of a structure than to return a structure and then use only one component of it. The compiler treats a function that returns a structure like a procedure, with an extra VAR parameter representing the result of the function. The function's caller allocates an unseen variable (on the stack) to receive the return value, but this "variable" is only allocated during execution of the statement that contains the function invocation.

## 12.5 Evaluating Expressions

In cases of ambiguity, an operator at a higher level is applied before one at a lower level. For instance, the following expression evaluates to 7 and not to 9:

```
1 + 2 * 3
```

Use parentheses to change operator precedence. Thus, the following evaluates to 9 rather than 7:

```
(1 + 2) * 3
```

If the $simple switch is on, sequences of operators of the same precedence are executed from left to right. If the switch is off, the compiler may rearrange expressions and evaluate common subexpressions only once, in order to generate optimized code. The semantics of the precedence relationships are retained, but normal associative and distributive laws are used. For example,

```
X * 3 + 12
```

is an optimization of:

```
3 * (6 + (X - 2))
```

These optimizations may occasionally give you unexpected overflow errors. For example,

```
(I - 100) + (J - 100)
```

will be optimized into the following:

```
(I + J) - 200
```

This may result in an overflow error, although the original expression did not (e.g., if "I" and "J" were each 16400).

An expression in your source file may or may not actually be evaluated when the program runs. For example, the expression F(X + Y) * 0 is always zero, so the subexpression (X + Y) and the function call need not be executed.

The compiler does not optimize real expressions as much as, for example, integer expressions, to make sure that the result of a real expression is always what a simple evaluation of the expression, as given, would be. For example, the integer expression

```
((1 + I) - 1) * J
```

is optimized to:

```
I * J
```

but the same expression with real variables is not optimized since the results may be different due to precision loss. Common subexpressions, such as 2 * X in SIN (2 * X) * COS (2 * X), may still be calculated just once and reloaded as necessary, but they are saved in a special 80-bit intermediate precision.

The order of evaluation may be fixed by parentheses:

```
(A + B) + C
```

is evaluated by adding A and B first, but

```
A + B + C
```

may be evaluated by adding A and B, B and C, or even A and C first.

Any expression can be passed as a CONST or CONSTS parameter or have its "address" found. The expression is calculated and stored in a temporary variable on the stack, and the address of this temporary variable can be used as a reference parameter or in some other address context.

To avoid ambiguities, enclose such an expression with operators or function calls in parentheses. For example, to invoke a procedure FOO (CONST X, Y : INTEGER ), FOO (I, (J + 14)) must be used instead of FOO (I, J + 14).

This implies a subtle distinction in the case of functions. For example:

```
FUNCTION SUM (CONST A, B : INTEGER) : INTEGER;
BEGIN
   SUM := A;
   IF B <> 0 THEN
     SUM := SUM (SUM, (SUM (B, 0) - 1)) + 1;
END;
```

In this example, SUM is called recursively, subtracting one from B until B is zero.

The use of a function identifier in a WITH statement follows a similar rule. For example, given a parameterless function, COMPLEX, which returns a record, "WITH COMPLEX" means "WITH the current value of the function." This can only occur inside the COMPLEX function itself. However, "WITH (COMPLEX)" causes the function to be called and the result assigned to a temporary local variable.

Another way to describe this is to distinguish between "address" and "value" phrases. The left-hand side of an assignment, a reference parameter, the ADR and ADS operators, and the WITH statement all need an address. The right-hand side of an assignment and a value parameter all need a value.

If an address is needed but only a value, such as a constant or an expression in parentheses, is available, the value must be put into memory so it has an address. For constants, the value goes in static memory; for expressions, the value goes in stack (local) memory. A function identifier refers to the current value of the function as an address, but causes the function to be called as a value.

Finally, in the scope of a function, the intrinsic procedure RESULT permits a reference to the current value of a function instead of invoking it recursively. For a function F, this means ADR F and ADR RESULT (F) are the same: the address of the current value of F. RESULT forces use of the current value in the same way that putting the function in parentheses, as in (F(X)), forces evaluation of the function.

## 12.6 Other Features of Expressions

EVAL and RESULT are two procedures available at the extend level for use with expressions. EVAL obtains the effect of a procedure from a function; RESULT yields the current value of a function within a function or nested procedure or function.

At the system level, the function RETYPE allows you to change the type of a value.

### 12.6.1 The EVAL Procedure

EVAL evaluates its parameters without actually calling anything. Generally, you use EVAL to obtain the effect of a procedure from a function. In such cases, the values returned by functions are of no interest, so EVAL is only useful for functions with side effects. For example, a function that advances to the next item and also returns the item might be called in EVAL just to advance to the next item, since there is no need to obtain a function return value.

Examples of the EVAL procedure:

```
EVAL (NEXTLABEL (TRUE))
EVAL (SIDEFUNC (X, Y), INDEX (4), COUNT)
```

### 12.6.2 The RESULT Function

Within the scope of a function, the intrinsic procedure RESULT permits a reference to the current value of a function instead of invoking it recursively. For a function F, this means ADR F and ADR RESULT (F) are the same; that is, the address of the current value of F. RESULT forces use of the current value in the same way that putting the function in parentheses as in (F (X)) forces evaluation of the function.

Examples of the RESULT function:

```
    FUNCTION FACTORIAL (I : INTEGER) : INTEGER;
    BEGIN
       FACTORIAL := 1; WHILE I > 1 DO
       BEGIN
          FACTORIAL := I * RESULT (FACTORIAL);
          I := I - 1
       END
    END;

   FUNCTION ABSVAL (I : INTEGER) : INTEGER;
   BEGIN
      ABSVAL := I;
      IF I < 0 THEN ABSVAL := -RESULT (ABSVAL)
   END;
```

## 12.6.3 The RETYPE Function

Occasionally, you need to change the type of a value. You can do this with the RETYPE function, available at the system level of MS-Pascal. If the new type is a structure, RETYPE can be followed by the usual selection syntax.

Examples of the RETYPE function:

```
RETYPE (COLOR, 3)   {inverse of ORD}
RETYPE (STRING2, I * J + K) [2]   {effect may vary}
```

You must use RETYPE with caution: it works on the memory byte level and ignores whether the low order byte of a two-byte number comes first or second in memory.

# Chapter 13
# Statements

The body of a program, procedure, or function contains statements. Statements denote actions that the program can execute. This chapter first discusses the syntax of statements and then separates and describes two categories of statements: simple statements and structured statements. A simple statement has no parts that are themselves other statements; a structured statement consists of two or more other statements. Table 13.1 lists the statements in each category in Microsoft Pascal.

**Table 13.1**

**Microsoft Pascal Statements**

| **Simple** | **Structured** |
|---|---|
| Assignment (:=) | Compound |
| Procedure | IF/THEN/ELSE |
| GOTO | CASE |
| BREAK | FOR |
| CYCLE | WHILE |
| RETURN | REPEAT |
| Empty | WITH |

## 13.1 The Syntax of Pascal Statements

Pascal statements are separated by a semicolon (;) and enclosed by reserved words such as BEGIN and END. A statement begins, optionally, with a label. Each of these three elements of statement syntax are discussed in the following sections.

### 13.1.1 Labels

Any statement referred to by a GOTO statement must have a label. A label at the standard level is one or more digits; leading zeros are ignored. Constant identifiers, expressions, and nondecimal notation cannot serve as labels.

All labels must be declared in a LABEL section. At the extend level, a label can also be an identifier.

Example using labels and GOTO statements:

```
PROGRAM LOOPS (INPUT, OUTPUT);
LABEL 1, HAWAII, MAINLAND;

BEGIN
   MAINLAND : GOTO 1;
   HAWAII : WRITELN ('Here I am in Hawaii');
   1 : GOTO HAWAII
END.
```

A loop label is any label immediately preceding a looping statement: WHILE, REPEAT, or FOR. At the extend level, a BREAK or CYCLE statement can also refer to a loop label.

Both a CASE constant list and a GOTO label may precede a statement, in which case the CASE constants come first and then the GOTO label. In the following example, 321 is a CASE value, 123 is a label:

```
321 : 123 : IF LOOP THEN GOTO 123
```

## 13.1.2 Separating Statements

Semicolons separate statements. Semicolons do not terminate statements. However, since Pascal permits the empty statement, using the semicolon as if it were a statement terminator is rarely disastrous.

Example showing semicolon to separate statements:

```
BEGIN
     10 : WRITELN;
     A := 2 + 3;
     GOTO 10
END
```

A common error is to terminate the THEN clause in an IF/THEN/ELSE statement with a semicolon. Thus, the following example generates a warning message:

```
IF A = 2 THEN WRITELN;
ELSE A = 3
```

Another common error is to put a semicolon after the DO in a WHILE or FOR statement:

```
FOR I := 1 TO 10 DO;
BEGIN
   A[I] := I;
   B[I] := 10 - I
END;
```

The previous example, as written, will "execute" an empty statement ten times, then execute the array assignments once. Since there are occasionally legitimate uses for repeating an empty statement, no warning is given when this occurs.

The semicolon also follows the reserved word END at the close of a block of program statements.

### 13.1.3 The Reserved Words BEGIN and END

Whenever you want a program to execute a group of statements, instead of a single simple statement, you may enclose the block with the reserved words BEGIN and END.

For example, the following group of statements between BEGIN and END will all be executed if the condition in the IF statement is TRUE:

```
IF (MAX > 10) THEN
BEGIN
   MAX = 10;
   MIN = 0;
   WRITELN (MAX, MIN)
END;
WRITELN ('done')
```

At the extend level, you may substitute a pair of square brackets for the pair of keywords BEGIN and END.

## 13.2 Simple Statements

A simple statement is one in which no part constitutes another statement. Simple statements in standard Pascal are:

1. the assignment statement
2. the procedure statement
3. the GOTO statement
4. the empty statement

The empty statement contains no symbols and denotes no action. It is included in the definition of the language primarily to permit you to use a semicolon after the last in a group of statements enclosed between BEGIN and END.

The extend level in MS-Pascal adds three simple statements: BREAK, CYCLE, and RETURN.

### 13.2.1 Assignment Statements

The assignment statement replaces the current value of a variable with a new value, which you specify as an expression. Assignment is denoted by an adjacent colon and equal sign character (:=).

Examples of assignment statements:

```
A := B

A[I] := 12 * 4 + (B * C)

X : = Y
{Illegal. Colon (:) and equal}
{sign (=) must be adjacent.}

A + 2 := B
{Illegal. A + 2 is not a variable.}

A := ADD (1,1)
```

The value of the expression must be assignment compatible with the type of the variable. Selection of the variable may involve indexing an array or dereferencing a pointer or address. If it does, the compiler may, depending on the optimizations performed, mix these actions with the evaluation of the expression. If the $simple metacommand is on, the expression is evaluated first.

An assignment to a nonlocal variable (including a function return) puts an equal sign (=) or percent sign (%) in the G column of the listing file. See Section 18.5, "Listing File Format," for more information about these and other symbols used in the listing.

Within the block of a function, an assignment to the identifier of the function sets the value returned by the function. The assignment to a function identifier may occur either within the actual body of the function or in the body of a procedure or function nested within it.

If the range checking switch is on, an assignment to a set, subrange, or LSTRING variable may imply a runtime call to the error checking code.

According to the MS-Pascal optimizer, each section of code without a label or other point that could receive control is eligible for rearrangement and common subexpression elimination. Naturally, the order of execution is retained when necessary.

Given these statements,

```
X := A + C + B;
Y := A + B;
Z := A
```

the compiler might generate code to perform the following operations:

1. Get the value of A and save it.
2. Add the value of B and save the result.
3. Add the value of C and assign it to X.
4. Assign the saved A + B value to Y.
5. Assign the saved A value to Z.

This optimization occurs only if assignment to X and Y and getting the value of A, B, or C are all independent. If C is a function without the PURE attribute and A is a global variable, evaluating C might change A. Then since the order of evaluation within an expression in this case is not fixed, the value of A in the first assignment could be the old value or the new one.

However, since the order of evaluation among statements is fixed, the value of A in the second and third assignments is the new value.

The following actions may limit the ability of the optimizer to find common subexpressions:

1. assignment to a nonlocal variable
2. assignment to a reference parameter
3. assignment to the referent of a pointer
4. assignment to the referent of an address variable
5. calling a procedure
6. calling a function without the PURE attribute

The optimizer does allow for "aliases," that is, a single variable with two identifiers, perhaps one as a global variable and one as a reference parameter.

### 13.2.2 Procedure Statements

A procedure statement executes the procedure denoted by the procedure identifier.

For example, assume you have defined the procedure DO_IT:

```
PROCEDURE DO_IT;
BEGIN
   WRITELN('Did it')
END;
```

DO_IT is now a statement that can be executed simply by invoking its name:

```
DO_IT
```

If you declare the procedure with a formal parameter list, the procedure statement must include the actual parameters.

MS-Pascal includes a large number of predeclared procedures. See Chapter 15, "Available Procedures and Functions," for complete information. One of the predeclared procedures is ASSIGN. You need not declare it in order to use it.

```
ASSIGN (INFILE, 'MYFILE')
```

Note that the ASSIGN procedure contains a parameter list. These parameters are the actual parameters that are bound to the formal parameters in the procedure declaration. For a discussion of formal and reference parameters, see Section 14.4, "Procedure and Function Parameters."

## 13.2.3 The GOTO Statement

A GOTO statement indicates that further processing continues at another part of the program text, namely at the place of the label. You must declare a LABEL in a LABEL declaration section, before using it in a GOTO statement.

Several restrictions apply to the use of GOTO statements:

1. A GOTO must not jump to a more deeply nested statement, that is, into an IF, CASE, WHILE, REPEAT, FOR, or WITH statement. GOTOs from one branch of an IF or CASE statement to another are permitted.
2. A GOTO from one procedure or function to a label in the main program or in a higher level procedure or function is permitted. A GOTO may jump out of one of these statements, so long as the statement is directly within the body of the procedure or function. However , such a jump generates extra code both at the location of the GOTO and at the location of the label. The GOTO and label must be in the same compiland, since labels, unlike variables, cannot be given the PUBLIC attribute.

Examples of GOTO statements, both legal and illegal:

```
PROGRAM LABEL_EXAMPLES;
LABEL 1, 2, 3, 4;
```

```
    PROCEDURE ONE;
    LABEL 11, 12, 13;

        PROCEDURE IN_ONE;
        LABEL 21;
        {Outer level GOTOs cannot jump into 21.}

        BEGIN
            IF TUESDAY THEN GOTO 1
            ELSE GOTO 11;
            {1 and 11 are both legal outer level labels.}
            21 : WRITE ('IN_ONE')
        END;

    BEGIN   {Procedure one}
        IF RAINING THEN GOTO 1 ELSE GOTO 11;
        {This is legal.}
        11 : GOTO 21
        {Illegal. Cannot jump into inner level}
        {procedures.}
    END;

    PROCEDURE TWO;
    BEGIN
        GOTO 11
        {Illegal. Cannot jump into different procedure}
        {at same level.}
    END;

BEGIN   {Main level}
    IF SEATTLE
    THEN
        BEGIN BEGIN
            GOTO 2;
            {OK to go to 2 at program level.}
            4 : WRITE ('here')
        END END
    ELSE GOTO 4;
    {OK to jump into THEN clause.}
    2 : GOTO 3;
    {Illegal. Cannot jump into REPEAT statement.}
    REPEAT
            WHILE MS_BYRON DO
                3 : GOTO 2
                {OK to jump out of loops.}
    UNTIL DATE;
    1 : GOTO 11
    {Illegal. Cannot jump into procedure from program.}
END.
```

If the $goto metacommand is on, every GOTO statement is flagged with a warning that reminds you that "GOTOs are considered harmful." This may be useful either in an educational environment or for finding all GOTOs in a program in order to locate a bug. The J (jumps) column of the listing file contains the following:

1. A plus (+) flags a GOTO to a label later in the listing.
2. A minus sign (-) marks a GOTO to a label already encountered in the listing.
3. An asterisk (*) flags a RETURN or a mixture of jumps.

See Section 18.5, "Listing File Format," for details about the listing file.

## 13.2.4 The BREAK, CYCLE, and RETURN Statements

At the extend level, BREAK, CYCLE, and RETURN statements are allowed in addition to the simple statements already described. These statements perform the following functions:

1. BREAK exits the currently executing loop.
2. CYCLE exits the current iteration of a loop and starts the next iteration.
3. RETURN exits the current procedure, function, program, or implementation.

All three statements are functionally equivalent to a GOTO statement.

1. A BREAK statement is a GOTO to the first statement after a repetitive statement.
2. A CYCLE statement is a GOTO to an implied empty statement after the body of a repetitive statement. This jump starts the next iteration of a loop. In either a WHILE or REPEAT statement, CYCLE performs the Boolean test in the WHILE or UNTIL clause before executing the statement again; in a FOR statement, CYCLE goes to the next value of the control variable.

3. A RETURN statement is a GOTO to an implied empty statement after the last statement in the current procedure or function or the body of a program or implementation.

The J (jump) column in the listing file contains a plus sign (+) for a BREAK or GOTO statement, a minus sign (-) for a CYCLE or GOTO statement, and an asterisk (*) for a RETURN statement or a mixture of jumps. See Section 18.5, "Listing File Format," for information about the listing file.

BREAK and CYCLE have two forms, one with a loop label and one without. If you give a loop label, the label identifies the loop to exit or restart. If you don't give a label, the innermost loop is assumed, as shown in the following example:

```
OUTER : FOR I := 1 TO N1 DO
    INNER : FOR J := 1 TO N2 DO
        IF A [I, J] = TARGET THEN BREAK OUTER;
```

## 13.3 Structured Statements

Structured statements are themselves composed of other statements. There are four kinds of structured statements:

1. compound statements
2. conditional statements
3. repetitive statements
4. WITH statements

The control level is shown in the the C (control) column of the listing file. The value in the C column is incremented each time control passes to a nested statement; conversely, this value is decremented each time control passes back to the nesting statement. This helps you search for a missing or extra END in a program.

## 13.3.1 Compound Statements

The compound statement is a sequence of simple statements, enclosed by the reserved words BEGIN and END. The components of a compound statement execute in the same sequence as they appear in the source file.

Examples of compound statements:

```
BEGIN
     TEMP := A [I];
     A[I] := A [J];
     A [J] := TEMP
     {Semicolon not needed here.}
END

BEGIN
     OPEN_DOOR;
     LET_EM_IN;
     CLOSE_DOOR;
     {Semicolon signifies empty statement.}
END
```

All MS-Pascal conditional and repetitive control structures (except REPEAT) operate on a single statement, not on multiple statements with ending delimiters. In this context, BEGIN and END serve as punctuation, like semicolon, colon, or parentheses. If you prefer, you may substitute a pair of square brackets for the BEGIN and END pair of reserved words. Note that a right bracket (]) matches only a left bracket ([) (not a BEGIN, CASE, or RECORD). In other words, a right bracket is not a synonym for END.

Brackets may not be used as synonyms for BEGIN and END to enclose the body of a program, implementation, procedure, or function; only BEGIN and END can be used for this purpose.

Examples of brackets replacing BEGIN and END:

```
IF FLAG THEN [X := 1; Y := -1]
ELSE [X := -1; Y := 0];

WHILE P.N <> NIL DO
     [Q := P; P := P.N; DISPOSE (Q)];

FUNCTION R2 (R : REAL) : REAL;
     [R2 := R * 2]
     {Illegal.}
```

## 13.3.2 Conditional Statements

A conditional statement selects for execution only one of its component statements. The conditional statements are the IF and CASE statements. Use the IF statement for one or two conditions, the CASE statement for multiple conditions.

### 13.3.2.1 The IF Statement

The IF statement allows for conditional execution of a statement. If the Boolean expression following the IF is true, the statement following the THEN is executed. If the Boolean expression following the IF is false, the statement following the ELSE, if present, is executed.

Examples of IF statements:

```
IF I > 0 THEN I := I - 1
{No semicolon here.}
ELSE I := I +1

IF (I <= TOP) AND (ARRI [I] <> TARGET) THEN
   I := I + 1

IF I <= TOP THEN
   IF ARRI [I] <> TARGET THEN
      I := I +1

IF I = 1 THEN
   IF J = 1 THEN
      WRITELN('I equals J')
   ELSE
      WRITELN('DONE only if I = 1 and J <> 1')
      {This ELSE is paired with the most deeply}
      {nested IF. Thus, the second WRITELN is}
      {executed only if I = 1 and J <> 1.}

IF I = 1 THEN BEGIN
   IF J = 1 THEN WRITELN('I equals J')
   END
ELSE
   WRITELN('DONE only if I <> 1')
      {Now the ELSE is paired with the first IF,}
      {since the second IF statement is}
      {bracketed by the BEGIN/END pair. Thus,}
      {the second WRITELN is executed if I <> 1.}
```

A semicolon (;) preceding an ELSE is always incorrect. The compiler skips it during compilation and issues a warning message.

The Boolean expression following an IF may include the sequential control operators described in Section 13.3.5, "Sequential Control," later in this chapter.

#### 13.3.2.2 The CASE Statement

The CASE statement consists of an expression (called the CASE index) and a list of statements. Each statement is preceded by a constant list, called a CASE constant list. The one statement executed is the one whose CASE constant list contains the current value of the CASE index. The CASE index and all constants must be of compatible, ordinal types.

Examples of CASE statements:

```
CASE OPERATOR OF
    PLUS : X := X + Y;
    MINUS : X := X - Y;
    TIMES: X := X * Y
END
{OPERATOR is the CASE index. PLUS, MINUS, and}
{TIMES are CASE constants. In this instance,}
{they are all of the values assumable by the}
{enumerated variable, OPERATOR.}

CASE NEXTCH OF
    'A' . . 'Z', '_' : IDENTIFIER;
    '+', '-', '*', '/' : OPERATOR;
    {Commas separate CASE constants}
    {and ranges of CASE constants.}
    OTHERWISE
        WRITE ('Unknown Character')
        {i.e., if any other character}
END
```

The CASE constant syntax is the same as for RECORD variant declarations. In standard Pascal, a CASE constant is one or more constants separated by commas. At the extend level, you may substitute a range of constants, such as 'A' . . 'Z', for a constant. No constant value can apply to more than one statement. The extend level also allows the CASE statement to end with an OTHERWISE clause. The OTHERWISE clause contains additional statements to be executed in the event that the CASE index value is not in the given set of CASE constant values. One of two things happens if the CASE index value is not in the set and no OTHERWISE clause is present:

1. If the range checking switch is on, a runtime error is generated.
2. If the range checking switch is off, the result is undefined (and may be catastrophic).

In MS-Pascal, control does not automatically pass to the next executable statement as in UCSD Pascal and some other languages. If you want this effect, include an empty OTHERWISE clause.

A semicolon (;) may appear after the final statement in the list, but is not required. The compiler skips over a colon (:) after an OTHERWISE and issues a warning.

Depending on optimization, the code generated by the compiler for a CASE statement may be either a "jump table" or series of comparisons (or both). If it is a jump table, a jump to an arbitrary location in memory can occur if the control variable is out of range and the range checking switch is off.

## 13.3.3 Repetition Statements

Repetition statements specify repeated execution of a statement. In standard Pascal, these include the WHILE, REPEAT, and FOR statements.

At the extend level in MS-Pascal, there are two additional statements, BREAK and CYCLE, for leaving or restarting the statements being repeated. These statements are functionally equivalent to a GOTO but easier to use.

### 13.3.3.1 The WHILE Statement

The WHILE statement repeats a statement zero or more times, until a Boolean expression becomes false.

Examples of WHILE statements:

```
WHILE P <> NIL DO P := NEXT (P)

WHILE NOT MICKEY DO
    BEGIN
        NEXTMOUSE;
        MICE : = MICE + 1
    END
```

The Boolean expression in a WHILE statement may include the sequential control operators described in Section 13.3.5, "Sequential Control."

Use WHILE if it is possible that no iterations of the loop may be necessary; use REPEAT where you expect that at least one iteration of the loop is required.

### 13.3.3.2 The REPEAT Statement

The REPEAT statement repeats a sequence of statements one or more times, until a Boolean expression becomes true.

Examples of REPEAT statements:

```
REPEAT
    READ (LINEBUFF);
    COUNT := COUNT + 1
UNTIL EOF;

REPEAT GAME UNTIL TIRED;
```

The Boolean expression in a REPEAT statement may include the sequential control operators described in Section 13.3.5, "Sequential Control."

Use the REPEAT statement to execute statements, not just a single statement, one or more times until a condition becomes true. This differs from the WHILE statement in which a single statement may not be executed at all.

### 13.3.3.3 The FOR Statement

The FOR statement tells the compiler to execute a statement repeatedly while a progression of values is assigned to a variable, called the control variable of the FOR statement. The values assigned start with a value called the initial value and end with one called the final value.

The FOR statement has two forms, one where the control variable increases in value and one where the control variable decreases in value:

```
FOR I := 1 TO 10 DO
{I is the control variable.}
  SUM := SUM + VICTORVECTOR [I]

FOR CH := 'Z' DOWNTO 'A' DO
{CH is the control variable.}
   WRITE (CH)
```

You may also use a FOR statement to step through the values of a set, as shown:

```
FOR TINT :=
      LOWER (SHADES) TO UPPER (SHADES) DO
             IF TINT IN SHADES
             THEN PAINT_AREA (TINT);
```

The ISO standard gives explicit rules regarding the control variable in FOR statements:

1. It must be of an ordinal type.
2. It must be an entire variable, not a component of a structure.
3. It must be local to the immediately enclosing program, procedure, or function and cannot be a reference parameter of the procedure or function.

   However, at the extend level of MS-Pascal, the control variable may also be any STATIC variable, such as a variable declared at the program level, unless the variable has a segmented ORIGIN attribute. Using a program level variable is an ISO error not caught.

4. No assignments to the control variable are allowed in the repeated statement. This error is caught by making the control variable READONLY within the FOR statement; it is not caught when a procedure or function invoked by the repeated statement alters the control variable. The control variable cannot be passed as a VAR (or VARS) parameter to a procedure or function.
5. The initial and final values of the control variable must be compatible with the type of the control variable. If the statement is executed, both the initial and final values must also be assignment compatible with the control variable. The initial value is always evaluated first, and then the final value. Both are evaluated only once before the statement executes.

The statement following the DO is not executed at all if:

1. The initial value is greater than the final value in the TO case.
2. The initial value is less than the final value in the DOWNTO case.

The sequence of values given the control variable starts with the initial value. This sequence is defined with the SUCC function for the TO case or the PRED function for the DOWNTO case until the last execution of the statement, when the control variable has its final value. The value of the control variable, after a FOR statement terminates naturally (whether or not the body executes), is undefined. It may vary due to optimization and, if $initck is on, may be set to an uninitialized value. However, the value of the control variable after leaving a FOR statement with GOTO or BREAK is defined as the value it had at the time of exit.

In standard Pascal, the body of a FOR statement may or may not be executed, so a test is necessary to see whether the body should be executed at all. However, if the control variable is of type WORD (or a subrange) and its initial value is a constant zero, the body must be executed no matter what the final value. In this case, no extra test need be executed and no code is generated to perform such a test.

Also, a control variable with the STATIC attribute may be more efficient than one that is not.

At the extend level in MS-Pascal, you may use temporary control variables:

```
FOR VAR control-variable
```

The prefix VAR causes the control variable to be declared local to the FOR statement (i.e., at a lower scope) and need not be declared in a VAR section. Such a control variable is not available outside the FOR statement, and any other variable with the same identifier is not available within the FOR statement itself. Other synonymous variables are, however, available to procedures or functions called within the FOR statement.

Examples of temporary control variables:

```
FOR VAR I := 1 TO 100 DO
    SUM := SUM + VICTOR [I]

FOR VAR COUNTDOWN := 10 DOWNTO LIFT_OFF DO
    MONITOR_ROCKET
```

### 13.3.3.4 The BREAK and CYCLE Statements

In theory, a program using the MS-Pascal extend level BREAK and CYCLE statements does not need to use any GOTO statements.

Each of these two statements has two forms, one with a loop label and one without. A loop label is a normal GOTO label prefixed to a FOR, WHILE, or REPEAT statement. Since, at the extend level, you may use identifier labels, a suggested practice is to use integers for labels referenced by GOTOs and identifiers for loop labels.

Examples of CYCLE and BREAK statements:

```
LABEL SEARCH, CLIMB;
   .
   .
SEARCH : WHILE I <= I_TOP DO
   IF PILE [I] = TARGET THEN BREAK SEARCH
   ELSE I := I + 1;
   .
   .
FOR I := 1 TO N DO
   IF NEXT [I] = NIL THEN BREAK;
   .
   .
CLIMB : WHILE NOT ITEM^.LEAF DO
   BEGIN
      IF ITEM^.LEFT <> NIL
        THEN [ITEM := ITEM^.LEFT; CYCLE CLIMB];
      IF ITEM^.RIGHT <> NIL
        THEN [ITEM := ITEM^.RIGHT; CYCLE CLIMB];
      WRITELN ('Very strange node');
      BREAK CLIMB
END;
```

### 13.3.4 The WITH Statement

The WITH statement opens the scope of a statement to include the fields of one or more records, so you can refer to the fields directly. For example, the following statements are equivalent:

```
WITH PERSON DO WRITE (NAME, ADDRESS, PHONE)
WRITE (PERSON.NAME, PERSON.ADDRESS, PERSON.PHONE)
```

The record given may be a variable, constant identifier, structured constant, or function identifier; it may not be a component of a PACKED structure. If you use a function identifier, it refers to the function's local result variable. If the record given in a WITH statement is a file buffer variable, the compiler issues a warning, since changing the position in the WITH statement may cause an error.

The record given can also be any expression in parentheses, in which case the expression is evaluated and the result assigned to a temporary (hidden) variable. If you want to evaluate a function designator, you must enclose it in parentheses.

You may give a list of records after the WITH, separated by commas. Each record so listed must be of a different type from all the others, since the field identifiers refer only to the last instance of the record with the type. These statements are equivalent:

```
WITH PMODE, QMODE DO statement
WITH PMODE DO WITH QMODE DO statement
```

Any record variable of a WITH statement that is a component of another variable is selected before the statement is executed. Active WITH variables should not be passed as VAR or VARS parameters, nor can their pointers be passed to the DISPOSE procedure. However, these errors are not caught by the compiler. Assignments to any of the record variables in the WITH list or components of these variables are allowed, as long as the WITH record is a variable.

In MS-Pascal, every WITH statement allocates an address variable that holds the address of the record. If the record variable is on the heap, the pointer to it should not be DISPOSEd within the WITH statement. If the record variable is a file buffer, no I/O should be done to the file within the WITH statement. Avoid assignments to the WITH record itself in programs intended to be portable.

## 13.3.5 Sequential Control

To increase execution speed or guarantee correct evaluation, it is often useful in IF, WHILE, and REPEAT statements to treat the Boolean expression as a series of tests. If one test fails, the remaining tests are not executed. Two extend level operators in MS-Pascal provide for such tests:

1. AND THEN

   X AND THEN Y is false if X is false; Y is only evaluated if X is true.

2. OR ELSE

   X OR ELSE Y is true if X is true; Y is only evaluated if X is false.

If you use several sequential control operators, the compiler evaluates them strictly from left to right.

Examples of sequential control operators:

```
IF SYM <> NIL AND THEN SYM^.VAL < 0 THEN
   NEXT_SYMBOL

WHILE I <= MAX AND THEN VECT [I] <> KEY DO
   I := I + 1;

REPEAT GEN (VAL)
UNTIL VAL = 0 OR ELSE (QU DIV VAL) = 0;

WHILE POOR AND THEN GETTING_POORER
   OR ELSE BROKE AND THEN BANKRUPT DO
        GET_RICH
```

You may only include these operators in the Boolean expression of an IF, WHILE, or UNTIL clause; they may not be used in other Boolean expressions. Furthermore, they may not occur in parentheses and are evaluated after all other operators.

# Chapter 14
# Introduction to Procedures and Functions

Procedures and functions act as subprograms that execute under the supervision of a main program. Unlike programs, however, procedures and functions can be nested within each other and can even call themselves. Furthermore, they have sophisticated parameter passing capabilities that programs lack.

Procedures are invoked as program statements; functions can be invoked in program statements wherever a value is called for.

The general format for procedures and functions is similar to the format for programs. The three-part structure includes a heading, declarations, and a body.

Example of a procedure declaration:

```
{Heading}
PROCEDURE MODEL (I : INTEGER; R : REAL);

{Beginning of declaration section}
LABEL  123;
CONST ATOP = 199;
TYPE INDEX = 0 . . ATOP;
VAR ARAY : ARRAY [INDEX] OF REAL; J : INDEX;
{Function declaration}
FUNCTION FONE (RX : REAL) : REAL;
BEGIN
    FONE := RX * I
END;

{Procedure declaration}
PROCEDURE FOUT (RY : REAL);
BEGIN
    WRITE ('Output is ', RY)
END;

{Body of procedure MODEL}
BEGIN
   FOR J := 0 TO ATOP DO
      IF GLOBALVAR THEN
      {Activation of procedure FOUT with}
      {value returned by function FONE.}
          FOUT (FONE (R + ARAY [J]))
        ELSE GOTO 123;
   123:    WRITELN ('Done')
END;
```

The declarations and body together are called the block.

The declaration of a procedure or function associates an identifier with a portion of a program. Later, you can activate that portion of the program with the appropriate procedure statement or function designator.

## 14.1 Procedures

The foregoing example illustrates the general format of a procedure declaration. The heading is followed by:

1. declarations for labels, constants, types, variables, and values
2. local procedures and functions
3. the body, which is enclosed by the reserved words BEGIN and END

When the body of a procedure finishes execution, control returns to the program element that called it.

At the standard level, the order of declarations must be as follows:

1. LABEL
2. CONST
3. TYPE
4. VAR
5. procedures and functions

At the extend level, you may have any number of LABEL, CONST, TYPE, VAR, and VALUE sections, as well as procedure and function declarations, in any order. Although data declarations (CONST, TYPE, VAR, VALUE) can be intermixed with procedure and function declarations, it is usually clearer to give all data declarations first.

However, putting variable declarations after procedure and function declarations guarantees that these variables will not be used by any of the procedures or functions.

In general, the initial values of variables are not defined. The VALUE section, which should follow the VAR section, is an MS-Pascal extension that lets you explicitly initialize program, module, implementation, STATIC, and PUBLIC variables. If the initialization switch ($initck) is on, all INTEGER, INTEGER subrange, REAL, and pointer variables are set to an uninitialized value. File variables are always initialized, regardless of the setting of the initialization switch.

## 14.2 Functions

Functions are the same as procedures, except that they are invoked in an expression instead of a statement and they return a value.

Function declarations define the parts of a program that compute a value. Functions are activated when a function designator, which is part of an expression, is evaluated.

A function declaration has the same format as a procedure declaration, except that the heading also gives the type of value returned by the function.

Example of a function heading:

```
    FUNCTION MAXIMUM (I, J : INTEGER) : INTEGER;
```

Within the block of a function, either in the body itself or in a procedure or function nested within the block, at least one assignment to the function identifier must be executed to set the return value. The compiler doesn't check for this assignment at runtime, unless the initialization switch is on and the returned type is INTEGER, REAL, or a pointer. However, if there is no assignment at all to the function identifier, the compiler issues an error message.

At the standard level, functions can return any simple type (ordinal, REAL, or INTEGER4) or a pointer. At the extend level, functions can return any simple, structured, or reference type.

However, they cannot return any type that cannot be assigned (i.e., a super array type or a structure containing a file, although a super array derived type is permitted).

A function identifier in an expression invokes the function recursively, rather than giving the current value of the function.

To obtain the current value, use the function RESULT, which takes the function identifier as a parameter and is available at the extend level.

The following is an example of a RESULT function used to obtain the current value of a function within an expression:

```
FUNCTION FACT (F : REAL) : REAL;
BEGIN
   FACT := 1;
   WHILE F > 1 DO
        BEGIN
           FACT := RESULT (FACT) * F; F := F - 1
        END
END;
```

Using the RESULT function is more efficient than using a separate local variable for the value of the function and then assigning this local variable to the function identifier before returning. If the function has a structured value, the usual component selection syntax can follow the RESULT function.

A function identifier on the left side of an assignment refers to the function's local variable, which contains its current value, instead of invoking the function recursively. Other places where using the function identifier refers to this local variable are the following:

1. a reference parameter
2. the record of a WITH statement

All of these uses involve getting the address (not the value) of a variable.

Instead of using the function's local variable, you may want to invoke the function and use the return value. As mentioned in Section 11.1, "What Is a Variable?" getting the address of an expression involves evaluating the expression, putting the resulting value into a temporary (hidden) variable, and using the address of this variable.

To do this for a function, you must force evaluation by putting the function designator in parentheses, as shown:

```
TYPE IREC = RECORD I : INTEGER END;

FUNCTION SUM (A, B : INTEGER) : IREC;
{Return sum of A and B.}
BEGIN
   IF TUESDAY THEN
   BEGIN                        {On Tuesdays, we recurse!}
      IF B = 0 THEN BEGIN SUM := A; RETURN END;
      WITH (SUM (A, B - 1))       {Call SUM recursively.}
         DO SUM.I := I + 1      {I is result of call.}
   END
   ELSE                         {Use function's}
      WITH SUM                  {local variable.}
         DO I := A + B          {I is local variable.}
END;
```

## 14.3 Attributes and Directives

An attribute gives additional information about a procedure or function. Attributes are available at the extend level of Microsoft Pascal. They are placed after the heading, enclosed in brackets, and separated by commas. Available attributes include ORIGIN, PUBLIC, FORTRAN, PURE, and INTERRUPT.

A directive gives information about a procedure or function, but it also indicates that only the heading of the procedure or function occurs, by replacing the block (declarations and body) normally included after the heading. Directives are available in standard Pascal. EXTERN and FORWARD are the only directives available. EXTERN can only be used with procedures or functions directly nested in a program, module, implementation, or interface. This restriction prevents access to nonlocal stack variables.

Table 14.1 displays the attributes and directives that apply to procedures and functions. Sections 14.3.1 through 14.3.7 describe these attributes and directives in detail.

**Table 14.1**

**Attributes and Directives for Procedures and Functions**

| Name | Purpose |
|---|---|
| FORWARD | A directive. Lets you call a procedure or function before you give its block in the source file. |
| EXTERN | A directive. Indicates that a procedure or function resides in another loaded module. |
| PUBLIC | An attribute. Indicates that a procedure or function may be accessed by other loaded modules. |
| ORIGIN | An attribute. Tells the compiler where the code for an EXTERN procedure or function resides. |
| FORTRAN | An attribute. Specifies a calling sequence for compatibility with Microsoft FORTRAN. |
| INTERRUPT | An attribute. Gives a procedure a special calling sequence that saves program status on the stack. |
| PURE | An attribute. Signifies that the function does not modify any global variables. |

The following rules apply when you combine attributes in the declaration of procedures and functions:

1. Any function may be given the PURE attribute.
2. Procedures and functions with attributes must be nested directly within a program, module, or unit. The only exception to this rule is the PURE attribute.
3. A given procedure or function may have only one calling sequence attribute (i.e., either FORTRAN or INTERRUPT, but not both).
4. PUBLIC and EXTERN are mutually exclusive, as are PUBLIC and ORIGIN.

The EXTERN or FORWARD directive is given automatically to all constituents of the interface of a unit; in the implementation, PUBLIC is given automatically to all constituents that are not EXTERN.

Since you declare the constituents of a unit only in the interface (not in the implementation), the interface is where you give the attributes. You may give the EXTERN directive in an implementation by declaring all EXTERN procedures and functions first; you may not use ORIGIN in either the interface or implementation of a unit.

In a module, you may give a group of attributes in the heading to apply to all directly nested procedures and functions. The only exception to this rule is the ORIGIN attribute, which may apply only to a single procedure or function.

If the PUBLIC attribute is one of a group of attributes in the heading of a module, an EXTERN attribute given to a procedure or function within the module explicitly overrides the global PUBLIC attribute. If the module heading has no attribute clause, the PUBLIC attribute is assumed for all directly nested procedures and functions.

The PUBLIC attribute allows a procedure or function to be called by other loaded code, and cannot be used with the EXTERN directive. The EXTERN directive permits a call to some other loaded code, using either the ORIGIN address or the linker. PUBLIC, EXTERN, and ORIGIN provide a low level way to link MS-Pascal routines with other routines in MS-Pascal or other languages.

A procedure or function declaration with the EXTERN or FORWARD directive consists only of the heading, without an enclosed block. EXTERN routines have an implied block outside of the program. FORWARD routines are fully declared (have a block) later in the same compiland. Both directives are available at the standard level of MS-Pascal. The keyword EXTERNAL is a synonym for EXTERN.

The PURE attribute applies only to functions, not to procedures. Conversely, INTERRUPT applies only to procedures, not to functions. PURE is the only attribute that can be used in nested functions.

## 14.3.1 The FORWARD Directive

A FORWARD declaration allows you to call a procedure or function before you fully declare it in the source text. This permits indirect recursion, where A calls B and B calls A. You make a FORWARD declaration by specifying a procedure or function heading, followed by the directive FORWARD. Later, you actually declare the procedure or function, without repeating the formal parameter list, attributes, or return types.

Example of a FORWARD declaration:

```
{Declaration of ALPHA, with parameter}
{list and attributes}
FUNCTION ALPHA (Q, R : REAL) : REAL [PUBLIC];
FORWARD;

{Call for ALPHA}
PROCEDURE BETA (VAR S, T : REAL);
BEGIN
   T := ALPHA (S, 3.14)
END;

{Actual declaration of ALPHA,}
{without parameter list}
FUNCTION ALPHA;
BEGIN
   ALPHA := (Q + R);
   IF R < 0.0 THEN BETA (3.14, ALPHA)
END;
```

## 14.3.2 The EXTERN Directive

The EXTERN directive identifies a procedure or function that resides in another loaded module. You give only the heading of the procedure or function, followed by the word EXTERN. The actual implementation of the procedure or function is presumed to exist in some other module.

EXTERN is an attribute when used with a variable, but a directive when used with a procedure or function. As with variables, the keyword EXTERNAL is a synonym for EXTERN.

The EXTERN directive for a particular procedure or function within a module overrides the PUBLIC attribute given for the entire module. The EXTERN directive is also permitted in an implementation of a unit for a constituent procedure or function. All such external constituents must be declared at the beginning of the implementation, before all other procedures and functions.

Any procedure or function with the EXTERN directive must be directly nested within a program. You can also link MS-Pascal routines by linking separately compiled units. See Chapter 17, "Compilable Parts of a Program."

Examples of procedure and function headings declared with the EXTERN directive:

```
FUNCTION POWER (X, Y : REAL) : REAL; EXTERN;

PROCEDURE ACCESS (KEY : KTYP) [ORIGIN SYSB + 4];
EXTERN;
```

In these examples, the function POWER is declared EXTERN, as is the procedure ACCESS. Both are implemented in external compilands. ACCESS also has the ORIGIN attribute, which is discussed later, in Section 14.3.4, "The ORIGIN Attribute."

You may not declare a procedure or function EXTERN if you have previously declared it FORWARD.

## 14.3.3 The PUBLIC Attribute

The PUBLIC attribute indicates a procedure or function that you can access from other loaded modules. In general, you access PUBLIC procedures and functions from other loaded modules by declaring them EXTERN in the modules that call them. Thus, you declare a procedure PUBLIC and define it in one module, and use it in another simply by declaring it EXTERN in the other module.

As with variables, the identifier of the procedure or function is passed to the linker, where it may be truncated if the linker requires it. PUBLIC and ORIGIN are mutually exclusive; PUBLIC routines need a following block, and ORIGIN routines must be EXTERN.

Any procedure or function with the PUBLIC attribute must be directly nested within a program or implementation. A higher level way to link MS-Pascal routines is by linking separately compiled units. See Chapter 17, "Compilable Parts of a Program," for details.

Examples of procedures and functions declared PUBLIC:

```
FUNCTION POWER (X, Y : REAL) : REAL [PUBLIC];
{The function POWER is available to other modules}
{because it has been declared PUBLIC.}
BEGIN
      .
      .
END;

PROCEDURE ACCESS (KEY : KTYP)
[ORIGIN SYSB + 4, PUBLIC];
BEGIN
      .
      .
END;
{Illegal since ORIGIN must also be EXTERN.}
```

## 14.3.4 The ORIGIN Attribute

The ORIGIN attribute must be used with the EXTERN directive; ORIGIN tells the compiler where the procedure or function can be found directly, so the linker does not require a corresponding PUBLIC identifier.

Examples of procedures and functions given the ORIGIN attribute:

```
PROCEDURE OPSYS [ORIGIN 8, FORTRAN];
EXTERN;

FUNCTION A_TO_D (C : SINT) : SINT [ORIGIN #100];
EXTERN;
```

In the first example, the procedure OPSYS begins at the absolute decimal address 8, has the FORTRAN calling sequence (described in Section 14.3.5, "The FORTRAN Attribute"), and is declared EXTERN. In the second example, the function A_TO_D takes a

SINT value as a parameter (SINT is the predeclared integer subrange from -127 to +127). The function is located at the hexadecimal address 100.

As with ORIGIN variables, the compiler uses the address to find the code and gives no directives to the linker. This permits, for example, calling routines at fixed addresses in ROM. In simple cases, it can substitute for a linking loader.

Remember that ORIGIN always implies EXTERN. Thus, procedures or functions that have previously been declared FORWARD cannot be declared with the ORIGIN attribute. Nor may you give ORIGIN as an attribute after the module heading.

Currently, you may not use the ORIGIN attribute with a constituent of a unit, either in an interface or in an implementation.

As with variables, the origin can be a segmented address. On segmented machines, a nonsegmented procedural origin assumes the current code segment with the offset given with the attribute.

## 14.3.5 The FORTRAN Attribute

The FORTRAN attribute applies both to procedures and functions (but not to variables). Instead of the usual Pascal calling sequence, it specifies a calling sequence that is compatible with the MS-FORTRAN compiler on your machine. This lets you call an MS-Pascal procedure or function from MS-FORTRAN programs and, conversely, external FORTRAN subroutines or MS-FORTRAN functions from an MS-Pascal program.

Example of a procedure with the FORTRAN attribute:

```
    PROCEDURE DELTA (I, J : INTEGER) [FORTRAN];
    FORWARD;
```

Any procedure or function with the FORTRAN attribute must be nested directly within a program or implementation.

In a 16-bit environment, MS-Pascal uses the same calling sequence as the compilers for Microsoft FORTRAN, Microsoft BASIC, and Microsoft COBOL. Thus, there is no need to give the FORTRAN attribute; if you do, it is ignored by the MS-Pascal Compiler.

## 14.3.6 The INTERRUPT Attribute

The INTERRUPT attribute applies only to procedures (not to functions or variables). It gives a procedure a special calling sequence that saves program status on the stack, which in turn allows a hardware interrupt to be processed, status restored, and control returned to the program, all without affecting the current state of the program.

Example of a procedure with the INTERRUPT attribute:

```
PROCEDURE INCHAR [INTERRUPT];
```

Because procedures with the INTERRUPT attribute are intended to be invoked by hardware interrupts, you may not invoke them with a procedure statement. An INTERRUPT procedure can only be invoked when the interrupt associated with it occurs. Furthermore, INTERRUPT procedures take no parameters.

Declaring a procedure with the INTERRUPT attribute ensures that the procedure conforms to the constraints of an interrupt handler in which:

1. a special calling sequence saves all status on the stack
2. the status saved includes machine registers and flags, plus any special global compiler data such as the frame pointer
3. the saved status is restored upon exit from the procedure

All INTERRUPT procedures must be nested directly within a compiland.

Interrupts are not automatically vectored to INERRUPT procedures; further, insofar as possible on the target machine, interrupts are neither enabled or disabled by an INTERRUPT procedure. Interrupt vectoring and enabling are too machine-dependent to be included in a machine-independent language like MS-Pascal.

However, MS-Pascal does provide the VECTIN library procedure, which takes an interrupt level and an interrupt procedure as parameters and sets the interrupt vector in a machine-dependent way.

Similarly, the library procedures ENABIN and DISBIN, respectively, enable and disable interrupts in a machine-dependent way. See Chapter 15, "Available Procedures and Functions," for more information on these routines.

An INTERRUPT procedure should usually return normally, in order to continue processing in the interrupted routine. This means the following:

1. You should not execute a GOTO that leaves an INTERRUPT procedure.
2. All debug checking should be turned off ($debug–, $entry–, and $runtime–).
3. Stack overflow may not be checked even if $stackck is on.

The use of INTERRUPT procedures introduces re-entrancy into MS-Pascal code: generated code is re-entrant, as is the runtime system (except for the heap unit and, in most operating systems, portions of the file unit).

Some critical sections in the runtime system are protected by semaphores that generate a runtime error if such a critical section is locked. For example, if the heap allocator is executing when an interrupt occurs and the INTERRUPT procedure tries to allocate a block from the heap, the structure of the heap could become invalid. This condition causes a runtime error.

However, in most cases, the file system is not protected by a semaphore. Therefore, it is safest to avoid performing any I/O within the INTERRUPT procedure. Alternatively, you can avoid most problems with I/O in an INTERRUPT procedure by not opening or closing any files (i.e., not declaring any local file variables or creating files on the heap) and by not performing input or output with any file that might be in the process of performing I/O when the interrupt occurs.

## 14.3.7 The PURE Attribute

The PURE attribute applies only to functions, not to procedures or variables. PURE indicates to the compiler's optimizer that the function does not modify any global variables either directly or by calling some other procedure or function.

Example of a PURE declaration:

```
FUNCTION AVERAGE (CONST TABLE : RVECTOR):
REAL [PURE];
```

For further illustration, examine these statements:

```
A := VEC [I * 10 + 7];
B := FOO;
C := VEC [I * 10 + 9];
```

If the function FOO is given the PURE attribute, the optimizer only generates code to compute I * 10 once. However, FOO, if it is not declared PURE, may modify I so that I * 10 must be recomputed after the call to FOO.

Functions are not considered PURE unless given the attribute explicitly. The compiler checks to see that a PURE function does not do any of the following:

1. assign to a nonlocal variable
2. have any VAR or VARS parameters (CONST and CONSTS parameters are permitted)
3. call any functions that are not PURE

Although the following additional restrictions are not checked, a PURE function should also:

1. not use the value of a global variable
2. not modify the referents of references passed by a value (e.g., pointer or address type referents)
3. not do input or output

Since the result of a PURE function with the same parameters must always be the same, the entire function call may be optimized away. For example, if in the following statements DSIN is PURE; the compiler only calls DSIN once:

```
HX := A * DSIN (P[I, J] * 2);
HY := B * DSIN (P[I, J] * 2);
```

## 14.4 Procedure and Function Parameters

Procedures and functions may take three different type of parameters:

1. value parameters
2. reference parameters
3. procedural and functional parameters

Each of these is discussed separately, in the order listed, in the following sections.

The discussion mentions both formal and actual parameters. A formal parameter is the parameter given when the procedure or function is declared, with an identifier in the heading. When the function or procedure is called, an actual parameter substitutes for the formal parameter given earlier; here the parameter takes the form of a variable or value or expression.

MS-Pascal has several parameter features at the extend level:

1. A super array type can be passed as a reference parameter.
2. A reference parameter can be declared READONLY.
3. Explicit segmented reference parameters can be declared.

## 14.4.1 Value Parameters

When a value parameter is passed, the actual parameter is an expression. That expression is evaluated in the scope of the calling procedure or function and assigned to the formal parameter. The formal parameter is a variable local to the procedure or function called. Thus, formal value parameters are always local to a procedure or function.

Example of value parameters:

```
{Function declaration}
FUNCTION ADD (A, B, C : REAL) : REAL;
    {A, B, and C are formal parameters}
  .
  .
  .
X := ADD (Y, ADD (1.111, 2.222, 3.333), (Z * 4))
```

In this particular function invocation, Y, ADD(. . .), and (Z * 4) are the expressions that make up the actual parameters. In this example, these expressions must all evaluate to the type REAL. (The example also recursively calls the function ADD.)

The actual parameter expression must be assignment compatible with the type of the formal parameter.

Passing structured types by value is permitted; however, it is inefficient, since the entire structure must be copied. A value parameter of a SET, LSTRING, or subrange type may also require a runtime error check if the range checking switch is on. In addition, SET and LSTRING value parameters may require extra generated code for size adjustment.

A file variable or super array variable cannot be passed as a value parameter, since it cannot be assigned. However, a variable with a type derived from a super array or file buffer variable can be passed. Passing a file buffer variable as a value parameter implies normal evaluation of the buffer variable.

## 14.4.2 Reference Parameters

When a reference parameter is passed at the standard level of MS-Pascal, the keyword VAR precedes the formal parameter. Furthermore, the actual parameter must be a variable, not an expression. The formal parameter denotes this actual variable during the execution of the procedure. Any operation on the formal parameter is performed immediately on the actual parameter, by passing the machine address of the actual variable to the procedure. For target processors with segmentation support, this address is an offset into the default data segment.

Example of variable parameters:

```
PROCEDURE CHANGE_VARS (VAR A, B, C : INTEGER);
{A, B, and C are formal reference parameters.}
{They denote variables, not values.}
  .
  .
CHANGE_VARS (X, Y, Z);
```

In this example, X, Y, and Z must be variables, not expressions. Also, the variables X, Y, and Z are altered whenever the formal parameters A, B, and C are altered in the declared procedure. This differs from the handling of value parameters, which can affect only the copies of values of variables. If the selection of the variable involves indexing an array or dereferencing a pointer or address, these actions are executed before the procedure itself. The type of the actual parameter must be identical to the type of the formal parameter.

Passing a nonlocal variable as a VAR parameter puts a slash (/) or percent sign (%) in the G (global) column of the listing file (see Section 18.5, "Listing File Format," for information about the significance of these characters in the G column of the listing).

None of the following may be passed as VAR parameters:

1. a component of a PACKED structure (except CHAR of a STRING or LSTRING)
2. any variable with a READONLY or PORT attribute (includes CONST and CONSTS parameters and the FOR control variable)

Passing a file buffer variable by reference generates a warning message, because it bypasses the normal file system call generated by the use of any buffer variable. These calls are not generated when a file variable is passed by reference.

On a segmented machine, a VAR parameter passes an address that is really an offset into a default data segment. In some cases, access to objects residing in other segments is required. To pass these objects by reference, you must tell the compiler to use a segmented address containing both segment register and offset values. The extend level includes the parameter prefix VARS instead of VAR:

```
    PROCEDURE CONCATS (VARS T, S : STRING);
```

You may only use VARS as a data parameter in procedures and functions, not in the declaration section of programs, procedures, and functions. VARS and CONSTS parameters are provided chiefly to maintain compatibility with machines that have two different size address spaces. These parameters are not necessary for a machine with a single size address space. On such machines, the reserved words VARS and CONSTS are equivalent to VAR and CONST.

#### 14.4.2.1 Super Array Parameters

Super array parameters may appear as formal reference parameters. This allows a procedure or function to operate on an array with a particular super array type (also a component type and index type), but without any fixed upper bounds. The formal parameter is a reference parameter of the super array type itself.

The actual parameter type must be a type derived from the super array type or the super array type itself (i.e., another reference parameter or dereferenced pointer). Except for comparing LSTRINGs, super array type parameters cannot be assigned or compared as a whole.

The actual upper and lower bounds of the array are available with the UPPER and LOWER functions; this permits routines that can operate on arrays of any size. An LSTRING actual parameter can be passed to a reference parameter of the super array type STRING. Therefore, the super array parameter STRING can be used for procedures and functions that operate on strings of both STRING and LSTRING types.

Example of super array parameters:

```
TYPE REALS = ARRAY [0 . . *] OF REAL;

PROCEDURE SUMRS (VAR X : REALS; CONST X : REALS);
BEGIN
    .
    .
END;
```

For more information, see the following sections of this manual:

Section 7.2, "Super Arrays"

Section 7.2.1, "STRINGs"

Section 7.2.2, "LSTRINGs"

### 14.4.2.2 Constant and Segment Parameters

At the extend level, a formal parameter preceded by the reserved word CONST implies that the actual parameter is a READONLY reference parameter. This is especially useful for parameters of structured types, which may be constants, since it eliminates the need for a time-consuming value parameter copy. The actual parameter can be a variable, function result, or constant value.

No assignments can be made to the CONST parameter or any of its components. CONST super array types are permitted. A CONST parameter in one procedure cannot be passed as a VAR parameter to another procedure. However, it is permissible to pass a VAR parameter in one procedure as a CONST parameter in another.

Example of a CONST parameter:

```
PROCEDURE ERROR (CONST ERRMSG : STRING);
```

On a segmented machine, a CONST parameter passes an address that is really an offset into a default data segment. In some cases, access to objects residing in other segments is required. To pass these objects by reference, you must tell the compiler to use a segmented address that contains both segment register and offset values. The extend level includes the parameter prefix CONSTS, instead of CONST. Use of CONSTS parameters parallels use of VARS for formal reference parameters.

Example of a CONSTS parameter:

```
PROCEDURE CAT (VARS T : STRING; CONSTS S : STRING);
```

A CONSTS parameter can only be used as a data parameter in procedures and functions, not in the declaration section of programs, procedures, and functions.

You can also pass the value of an expression as a CONST or CONSTS parameter. The expression is evaluated and assigned to a temporary (hidden) variable in the frame of the calling procedure or function. You should enclose such an expression in parentheses to force its evaluation.

A function identifier can be passed by reference as a VAR, VARS, CONST, or CONSTS parameter. The function's local variable is passed, so the call must occur in the function's body or in a procedure or function declared with the function.

The value returned by a function designator can also be passed, like any expression, as a CONST or CONSTS parameter. Like any expression passed by reference, the function designator should be enclosed in parentheses, as shown:

```
PROCEDURE WRITE_ANSWER (CONSTS A : INTEGER);
BEGIN
    WRITELN ('THE ANSWER IS ,' A)
END;

FUNCTION ANSWER : INTEGER;
BEGIN
    ANSWER := 42;
    WRITE_ANSWER (ANSWER);
    {Pass reference to local variable.}
END;

PROCEDURE HITCH_HIKE;
BEGIN
    WRITE_ANSWER ((ANSWER))
    {Call ANSWER, assign to temporary variable,}
    {pass reference to temporary variable.}
END;
```

## 14.4.3 Procedural and Functional Parameters

Procedural parameters can be used in the following circumstances:

1. in numerical analysis
2. in calling some library routines
3. in special applications

In numerical analysis, you might pass a function to a procedure or function that finds an integral between limits, a maximum or minimum value, and so on. Some interesting algorithms in areas such as parsing and artificial intelligence also use procedural parameters.

When a procedural or functional parameter is passed, the actual identifier is that for a procedure or function. The formal parameter is a procedure or function heading, including any attributes, preceded by the reserved word PROCEDURE or FUNCTION.

For example, examine these declarations:

```
TYPE DOOR  = (FRONT, BARN, CELL, DOG_HOUSE);
     SPEED = (FAST, SLOW, NORMAL);
     DIRECTION = (OPEN, SHUT);

PROCEDURE OPEN_DOOR_WIDE
    (VAR A : DOOR; B : SPEED; C : DIRECTION);
     .
     .
PROCEDURE SLAM_DOOR
    (VAR DR : DOOR; SP : SPEED; DIR : DIRECTION);
     .
     .
PROCEDURE LEAVE_AJAR
    (VAR DD : DOOR; SS : SPEED; DD : DIRECTION);
```

All of the procedures in the example have parameter lists of equal length. The types of the parameters are not only compatible, but also identical. The formal parameters need not be identically named.

A procedural or functional parameter can accept one of these procedures if the procedure or function is set up correctly, as shown:

```
FUNCTION DOOR_STATUS (PROCEDURE MOVE_DOOR)
     (VAR  X : DOOR; Y : SPEED; Z : DIRECTION);
     (VAR XX : DOOR; YY : SPEED; ZZ : DIRECTION) : INTEGER;
     {"PROCEDURE MOVE_DOOR" is the formal procedural}
     {parameter; the next two lines are other formal}
     {parameters.}

BEGIN {door_status}
     DOOR_STATUS := 0;
     MOVE_DOOR (XX, YY, ZZ);
     {One of the three procedures declared}
     {previously is executed here.}

     IF  XX = BARN AND ZZ = SHUT
     THEN DOOR_STATUS := 1;

     IF  XX = CELL AND ZZ = OPEN
     THEN DOOR_STATUS := 2;

     IF  XX = DOG_HOUSE AND ZZ = SHUT
     THEN DOOR_STATUS := 3
END;
```

Use of the procedural parameter MOVEDOOR might occur in program statements as follows:

```
IF DOOR_STATUS
     (SLAM_DOOR, CELL, FAST, SHUT) = 0
THEN
     SOCIETY := SAFE;
IF DOOR_STATUS
     (OPEN_DOOR_WIDE, BARN, SLOW, OPEN) = 0
THEN
     COWS_ARE_OUT := TRUE;
IF DOOR_STATUS
     (LEAVE_AJAR, DOG_HOUSE, SLOW, OPEN) = 0
THEN
     DOG_CAN_GET_IN := TRUE;
```

In each case above, the actual procedure list is compatible with the formal list, both in number and in type of parameters. If the parameter passed were a functional parameter, then the function return value would also have to be of an identical type.

In addition, the set of attributes for both the formal and actual procedural type must be the same, except that the PUBLIC and ORIGIN attributes and EXTERN directive are ignored.

A PUBLIC or EXTERN procedure, or any local procedure at any nesting level, can be passed to the same type of formal parameter. However, the PURE attribute and any calling sequence attributes must match. Also, in systems with segmented code addresses, a procedure or function passed as a parameter to an EXTERN procedure or function must itself be PUBLIC or EXTERN.

In MS-Pascal, you cannot pass predeclared procedures and functions compiled as in-line code; you can only pass them in called subroutines. Also, the READ, WRITE, ENCODE, and DECODE families are translated into other calls by the compiler, based on the argument types, and so cannot be passed. Corresponding routines in the file unit or encode/decode unit can be passed, however. For example, a READ of an INTEGER becomes a call to RTIFQQ and this procedure can be passed as a parameter.

The following intrinsic procedures and functions cannot be passed as procedure or function parameters:

1. at the standard level of MS-Pascal

| | | | |
|---|---|---|---|
| ABS | EOLN | PACK | SQR |
| ARCTAN | EXP | PAGE | SQRT |
| CHR | LN | PRED | SUCC |
| COS | NEW | READ | UNPACK |
| DISPOSE | ODD | READLN | WRITE |
| EOF | ORD | SIN | WRITELN |

2. at the extend and system levels of MS-Pascal

| | | | |
|---|---|---|---|
| BYLONG | FLOAT4 | READFN | SIZEOF |
| BYWORD | HIBYTE | READSET | TRUNC |
| DECODE | HIWORD | RESULT | TRUNC4 |
| ENCODE | LOBYTE | RETYPE | UPPER |
| EVAL | LOWER | ROUND | WRD |
| FLOAT | LOWORD | ROUND4 | |

When a procedure or function passed as a parameter is finally activated, any nonlocal variables accessed are those in effect at the time the procedure or function is passed as a parameter, rather than those in effect when it is activated. Internally, both the address of the routine and the address of the upper frame (in the stack) are passed.

Example of formal procedure use:

```
PROCEDURE ALPHA;
   VAR I : INTEGER;

   PROCEDURE DELTA;
      BEGIN
         WRITELN ('Delta done')
      END;

   PROCEDURE BETA (PROCEDURE XPR);
      VAR GLOB : INTEGER;

      PROCEDURE GAMMA;
         BEGIN GLOB := GLOB + 1 END;
      BEGIN {Start BETA}
         GLOB := 0;
         IF I = 0
           THEN BEGIN
                I := 1; XPR; BETA (GAMMA)
                END
           ELSE BEGIN
                GLOB := GLOB + 1; XPR
                END
      END;

   BEGIN {Start ALPHA}
      I := 0;
      BETA (DELTA)
   END;
```

The following list describes what happens in this example:

1. ALPHA is called.
2. BETA is called, passing the procedure DELTA.
3. This latter call creates an instance of GLOB on the stack (call it GLOB1).
4. BETA first clears GLOB1 by setting it to zero. Then, since I is 0, the THEN clause is executed, which sets I to one and executes XPR, which is bound to DELTA.
5. Therefore, 'Delta done' is written to OUTPUT.

6. Now BETA is called recursively. BETA is passed the address of GAMMA, and, at this time, the access path to any nonlocal variables used by GAMMA (i.e., GLOB1) is passed as well.

7. The second call to BETA creates another instance of GLOB (GLOB2). When GLOB2 is cleared this time, I is 1, so GLOB2 is incremented.

8. Then XPR is called, which is bound to GAMMA, so GAMMA is executed and increments the instance of GLOB active when GAMMA was passed to BETA, GLOB1.

9. GAMMA returns, the second BETA call returns, the first BETA call returns, and finally, ALPHA returns.

# Chapter 15
# Available Procedures and Functions

All versions of Pascal predeclare a large number of common procedures and functions. You do not have to declare these procedures and functions in a program. Since they are defined in a scope "outside" the program, you may redefine these identifiers.

To promote portability, Microsoft Pascal makes some of the predeclared procedures and functions available only at the extend or at the system level. MS-Pascal also includes some useful library procedures and functions that you must declare EXTERN in order to use.

MS-Pascal implements three kinds of procedures and functions:

1. Some are predeclared, and the compiler translates them into other calls or special generated code (these you cannot pass as parameters).
2. Some are predeclared but you call them normally (except for a name change).
3. Some are not predeclared but are available as part of the MS-Pascal runtime library (these you must declare explicitly).

However, it is more useful when discussing these procedures and functions to categorize them by what they do rather than by how they are implemented. Table 15.1 shows this categorization.

**Table 15.1**

**Categories of Available Procedures and Functions**

| Category | Purpose |
|---|---|
| File system | Operate on files of different modes and structures |
| Dynamic allocation | Dynamically allocate and deallocate data structures on the heap at runtime |
| Data conversion | Convert data from one type to another |
| Arithmetic | Perform common transcendental and other numeric functions |
| Extend level intrinsics | Provide additional procedures and functions at the extend level of MS-Pascal |
| System level intrinsics | Provide additional procedures and functions at the system level of MS-Pascal |
| String intrinsics | Operate on STRING and LSTRING type data |
| Library | Available in the MS-Pascal runtime library: they are not predeclared; you must declare them with the EXTERN directive |

## 15.1 Categories of Available Procedures and Functions

The rest of this chapter is divided into two sections. Section 15.1 describes each of the categories shown in the preceding table and lists the procedures and functions each category includes. Section 15.2 is an alphabetical directory of all of the available procedures and functions. Each entry includes the syntax and a description, plus examples and notes as appropriate.

### 15.1.1 File System Procedures and Functions

The MS-Pascal file system supports a variety of procedures and functions that operate on files of different modes and structures.

These procedures and functions fall into three categories, as shown in Table 15.2.

**Table 15.2**

**File System Procedures and Functions**

| Category | Procedures | Functions |
|---|---|---|
| Primitive | GET<br>PAGE<br>PUT<br>RESET<br>REWRITE | EOF<br>EOLN |
| Textfile I/O | READ<br>READLN<br>WRITE<br>WRITELN | |
| Extend Level I/O | ASSIGN<br>CLOSE<br>DISCARD<br>READSET<br>READFN<br>SEEK | |

For details on each of these procedures and functions, see Chapter 16, "File-Oriented Procedures and Functions."

## 15.1.2 Dynamic Allocation Procedures

Two procedures, NEW and DISPOSE, allow dynamic allocation and deallocation of data structures at runtime. NEW allocates a variable in the heap, and DISPOSE releases it.

## 15.1.3 Data Conversion Procedures and Functions

Use the following procedures and functions to convert data from one type to another:

| | | |
|---|---|---|
| CHR | PACK | TRUNC |
| FLOAT | PRED | TRUNC4 |
| FLOAT4 | ROUND | UNPACK |
| ODD | ROUND4 | WRD |
| ORD | SUCC | |

Four of these convert any ordinal type to a particular ordinal type:

1. CHR (ordinal) to CHAR
2. ODD (ordinal) to BOOLEAN
3. ORD (ordinal) to INTEGER
4. WRD (ordinal) to WORD

PRED and SUCC also operate on ordinal types.

Six of the conversion procedures and functions convert between INTEGER or INTEGER4 and REAL:

1. FLOAT converts INTEGER to REAL
2. FLOAT4 converts INTEGER4 to REAL
3. ROUND converts REAL to INTEGER
4. ROUND4 converts REAL to INTEGER4
5. TRUNC converts REAL to INTEGER
6. TRUNC4 converts REAL to INTEGER4

PACK and UNPACK transfer components between packed and unpacked arrays.

## 15.1.4 Arithmetic Functions

All arithmetic functions take a CONSTS parameter of type REAL4 or REAL8, or a type compatible with INTEGER (labeled "numeric" in the directory). ABS and SQR also take WORD and INTEGER4 values.

All functions on REAL data types check for an invalid (uninitialized) value. They also check for particular error conditions and generate a runtime error message if an error condition is found.

If the math checking switch is on, errors in the use of the functions ABS and SQR on INTEGER, WORD, and INTEGER4 data generate a runtime error message. If the switch is off, the result of an error is undefined.

Table 15.3 lists the arithmetic functions available, along with the routines called depending on whether single or double precision is required.

**Table 15.3**

**Predeclared Arithmetic Functions**

| Name | Operation | REAL4 | REAL8 |
|---|---|---|---|
| ABS | Absolute value | (inline) | (inline) |
| ARCTAN | Arc tangent | ATSRQQ | ATDRQQ |
| COS | Cosine | CNSRQQ | CNDRQQ |
| EXP | Exponential | EXSRQQ | EXDRQQ |
| LN | Natural log | LNSRQQ | LNDRQQ |
| SIN | Sine | SNSRQQ | SNDRQQ |
| SQR | Square | (inline) | (inline) |
| SQRT | Square root | SRSRQQ | SRDRQQ |

The MS-FORTRAN runtime library provides several additional REAL4 and REAL8 functions, as shown in Table 15.4. If you use them, you must declare them with the EXTERN directive.

## Table 15.4

## REAL Functions from the Microsoft FORTRAN Runtime Library

| Operation | REAL4 | REAL8 |
|---|---|---|
| Arc cosine | ACSRQQ | ACDRQQ |
| Integral trunc | AISRQQ | AIDRQQ |
| Integral round | ANSRQQ | ANDRQQ |
| Arc sine | ASSRQQ | ASDRQQ |
| Arc tangent A/B | A2SRQQ | A2DRQQ |
| Hyperbolic cosine | CHSRQQ | CHDRQQ |
| Decimal log | LDSRQQ | LDDRQQ |
| Modulo | MDSRQQ | MDDRQQ |
| Minimum | MNSRQQ | MNDRQQ |
| Maximum | MXSRQQ | MXDRQQ |
| Power (REAL8**INTG4) | | PIDRQQ |
| Power (REAL4**INTG4) | PISRQQ | |
| Power (REAL**REAL) | PRSRQQ | PRDRQQ |
| Hyperbolic sine | SHSRQQ | SHDRQQ |
| Hyperbolic tangent | THSRQQ | THDRQQ |
| Tangent | TNSRQQ | TNDRQQ |

Some common mathematical functions are not standard in Pascal, but are relatively simple to accomplish with program statements or to define as functions in a program. Some typical definitions follow:

```
SIGN (X) is ORD (X > 0) - ORD (X < 0)
POWER (X, Y) is EXP (Y * LN (X))
```

You could also write your own functions in MS-Pascal to do the same thing. Defining functions like these is a good opportunity to use the PURE attribute (to obtain more efficient code). For example:

```
FUNCTION POWER (A, B : REAL) : REAL [PURE];
BEGIN
   IF A <= 0 THEN
      ABORT ('Nonplus real to power', 24, 0);
      POWER := EXP (B * LN (A))
END;
```

## 15.1.5 Exend Level Intrinsics

At the extend level of MS-Pascal, the following intrinsic procedures and functions are available:

| | | |
|---|---|---|
| ABORT | EVAL | LOWORD |
| BYLONG | HIBYTE | RESULT |
| BYWORD | HIWORD | SIZEOF |
| DECODE | LOBYTE | UPPER |
| ENCODE | LOWER | |

Several of these are used to compose and decompose one-byte, two-byte, and four-byte items: HIBYTE, LOBYTE, BYWORD, HIWORD, LOWORD, and BYLONG.

ENCODE and DECODE convert between internal and string forms of variables. ABORT invokes a runtime error.

The others, EVAL, LOWER, UPPER, RESULT, and SIZEOF, are used in special situations (described for each function in Section 15.2, "Directory of Functions and Procedures").

## 15.1.6 System Level Intrinsics

Several additional intrinsic procedures and functions are available at the system level:

| | |
|---|---|
| FILLC | MOVESL |
| FILLSC | MOVESR |
| MOVEL | RETYPE |
| MOVER | |

The MOVE and FILL procedures perform low-level operations on byte strings. RETYPE changes the type of an expression arbitrarily.

## 15.1.7 String Intrinsics

The string intrinsics feature provides a set of procedures and functions, some of which operate on STRINGs and LSTRINGs, and some on LSTRINGs only as shown in Table 15.5:

**Table 15.5**

**String Procedures and Functions**

| Name | Parameter |
|---|---|
| CONCAT | STRING |
| DELETE | STRING |
| INSERT | STRING |
| COPYLST | STRING |
| COPYSTR | STRING or LSTRING |
| POSITN | STRING or LSTRING |
| SCANEQ | STRING or LSTRING |
| SCANNE | STRING or LSTRING |

## 15.1.8 Library Procedures and Functions

The following routines are not predeclared, but are available to you in the MS-Pascal runtime library. You must declare them, with the EXTERN directive, before using them in a program.

1. Initialization and termination routines

   BEGOQQ and ENDOQQ are called during initialization and termination, respectively. You might use them to invoke a debugger or to write customized messages, such as the time of execution, to the terminal screen.

2. Heap management routines

   Heap management routines complement the standard NEW and DISPOSE procedures and include:

   ALLHQQ

   FREECT

   MARKAS

   MEMAVL

   RELEAS

3. Interrupt routines

   These routines handle interrupt processing, although the actual effect varies with the target machine:

   ENABIN

   DISBIN

   VECTIN

4. Terminal I/O routines

   The following routines support direct input to and output from your terminal:

   GTYUQQ

   PTYUQQ

   PLYUQQ

5. Semaphore routines

   The two procedures, LOCKED and UNLOCK, provide a binary semaphore capability. You can use them to ensure exclusive access of a resource in a concurrent system.

6. No-overflow arithmetic functions

   These functions implement 16-bit and 32-bit modulo arithmetic. Overflow or carry is returned, instead of invoking a runtime error.

   LADDOK

   LMULOK

   SADDOK

   SMULOK

   UADDOK

   UMULOK

7. Clock routines

   These provide operating system clock information:

   TIME

   DATE

   TICS

## 15.2 Directory of Functions and Procedures

This section contains a list of all available procedures and functions, both those that are predeclared and those library routines that may be used if declared EXTERN. Each entry includes the heading, the category to which the operation belongs, and a description of what the procedure or function does. Notes and examples are included as appropriate. The headings given are the same for both REAL4 or REAL8, unless specifically stated otherwise.

**PROCEDURE ABORT (CONST STRING, WORD, WORD);**

An extend level intrinsic procedure. Halts program execution in the same way as an internal runtime error. The STRING (or LSTRING) is an error message. The string parameter is a CONST, not a CONSTS parameter. The first WORD is an error code (see Appendix H, "Messages," for error code allocations); the second WORD can be anything. The second WORD is sometimes used to return a file error status code from the operating system.

The parameters, as well as any information about the machine state (program counter, frame pointer, stack pointer) and the source position of the ABORT call (if the $line and/or $entry debugging switches are on), are given to you in a termination message or are available to the debugging package.

If the $runtime switch is on, then error messages report the location of the procedure or function that has called the routine in which abort was called. If $runtime is on, $line and $entry should be off, and routines in a source file should only call other $runtime routines.

**FUNCTION ABS (X : NUMERIC) : NUMERIC;**

An arithmetic function. Returns the absolute value of X. Both X and the return value are of the same numeric type: REAL4, REAL8, INTEGER, WORD, or INTEGER4. Since WORD values are unsigned, ABS (X) always returns X if X is of type WORD.

**FUNCTION ACSRQQ (CONSTS A : REAL4) : REAL4;**
**FUNCTION ACDRQQ (CONSTS A : REAL8) : REAL8;**

Arithmetic functions. Return the arc cosine of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**FUNCTION AISRQQ (CONSTS A : REAL4) : REAL4;**
**FUNCTION AIDRQQ (CONSTS A : REAL8) : REAL8;**

Arithmetic functions. Return the integral part of A, truncated toward zero. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**FUNCTION ALLHQQ (SIZE : WORD) : WORD;**

A library routine (heap management function). Returns zero if the heap is full, one if the heap structure is in error, or MAXWORD if the allocator has been interrupted. Otherwise, it returns the pointer value for an allocated variable with the size requested.

Generally, you use ALLHQQ with the RETYPE function. For example:

```
P_VAR := RETYPE (P_TYPE, ALLHQQ (28));
{RETYPE converts the value returned by}
{ALLHQQ (28) to the type P_TYPE.}
{This value is assigned to P_VAR.}

IF WRD (P_VAR) < 2 THEN GO_ABORT;
{PVAR is then checked for a heap}
{full or heap structure error.}
```

**FUNCTION ALLMQQ (WANTS : WORD) : ADSMEM;**

A library routine (segmented heap management function). This function returns a long segmented address of type ADSMEM. ALLMQQ takes a parameter "wants" that is a memory request in bytes.

A number "0" in the ".r" field of the segmented address returned by ALLMQQ indicates that memory of the requested "wants" size could not be allocated. A number "1" indicates that the long segmented heap is invalid.

ALLMQQ may not be available in your implementation of Microsoft Pascal.

**FUNCTION ANSRQQ (CONSTS A : REAL4) : REAL4;**
**FUNCTION ANDRQQ (CONSTS A : REAL8) : REAL8;**

Arithmetic functions. Like AISRQQ and AIDRQQ, return the truncated integral part of A, but round away from zero. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**FUNCTION ARCTAN (X : REAL) : REAL;**

An arithmetic function. Returns the arc tangent of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare ATSRQQ (CONSTS REAL4) and/or ATDRQQ (CONSTS REAL8) and use them instead.

**FUNCTION ASSRQQ (CONSTS A : REAL4) : REAL4;**
**FUNCTION ASDRQQ (CONSTS A : REAL8) : REAL8;**

Arithmetic functions. Return the arc sine of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**PROCEDURE ASSIGN (VAR F; CONSTS N : STRING);**

A file system procedure (extend level I/O). Assigns an operating system filename in a STRING (or LSTRING) to a file F.

See Section 16.3.1, "Extend Level Procedures," for a description of ASSIGN.

**FUNCTION ATSRQQ (CONSTS A : REAL4) : REAL4;**
**FUNCTION ATDRQQ (CONSTS A : REAL8) : REAL8;**

See FUNCTION ARCTAN.

**FUNCTION A2SRQQ (A, B : REAL4) : REAL4;**
**FUNCTION A2DRQQ (A, B : REAL8) : REAL8;**

Arithmetic functions. Return the arc tangent of (A/B). Both A and B, as well as the return value, are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**PROCEDURE BEGOQQ;**

A library routine (initialization). BEGOQQ is called during initialization, and the default version does nothing. However, you may write your own version of BEGOQQ, if you want, to invoke a debugger or to write customized messages, such as the time of execution, to a terminal screen.

See also PROCEDURE ENDOQQ.

**PROCEDURE BEGXQQ;**

A library routine (initialization). After your program is linked and loaded, BEGXQQ is the defined entry point for the load module.

As the overall initialization routine, BEGXQQ performs the following actions:

1. It resets the stack and the heap.
2. It initializes the file system.
3. It calls BEGOQQ.
4. It calls the program body.

BEGXQQ may be useful for restarting after a catastrophic error in a ROM-based system. However, invoking this procedure to restart a program does not take care of closing any files that may have previously been opened. Similarly, it does not re-initialize variables originally set in a VALUE section or with the initialization switch on.

## FUNCTION BYLONG (INTEGER-WORD, INTEGER-WORD) : INTEGER4;

An extend level intrinsic function. Converts WORDs or INTEGERs (or the LOWORDs of INTEGER4s) to an INTEGER4 value. BYLONG concatenates its operands:

```
BYLONG (A, B) =
ORD (LOWORD (A)) * 65535 + WRD (HIWORD (B))
```

If the first value is of type WORD, its most significant bit becomes the sign of the result.

## FUNCTION BYWORD (ONE-BYTE, ONE-BYTE) : WORD;

An extend level intrinsic function. Converts bytes (or the LOBYTEs of INTEGERs or WORDs) to a WORD value. Takes two parameters of any ordinal type. BYWORD returns a WORD with the first byte in the most significant part and the second byte in the least significant part:

```
BYWORD (A, B) = LOBYTE(A) * 256 + LOBYTE(B)
```

If the first value is of type WORD, its most significant bit becomes the sign of the result.

**FUNCTION CHR (X : ORDINAL) : CHAR;**

A data conversion function. Converts any ordinal type to CHAR. The ASCII code for the result is ORD (X). This is an extension to the ISO standard, which requires X to be of type INTEGER. An error occurs if ORD (X) > 255 or ORD (X) < 0. However, the error is caught only if the range checking switch is on.

**FUNCTION CHSRQQ (CONSTS A : REAL4) : REAL4;**
**FUNCTION CHDRQQ (CONSTS A : REAL8) : REAL8;**

Arithmetic functions. Return the hyperbolic cosine of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**PROCEDURE CLOSE (VAR F);**

A file system procedure (extend level I/O). Performs an operating system close on a file, ensuring that the file access is terminated correctly.

See Section 16.3.1, "Extend Level Procedures," for a description of CLOSE.

**FUNCTION CNSRQQ (CONSTS A : REAL4) : REAL4;**
**FUNCTION CNDRQQ (CONSTS A : REAL4) : REAL4;**

See FUNCTION COS.

**PROCEDURE CONCAT (VARS D : LSTRING;**
**CONSTS S : STRING);**

A string intrinsic procedure. Concatenates S to the end of D. The length of D increases by the length of S. An error occurs if D is too small, i.e., if UPPER (D) < D.LEN + UPPER (S).

**PROCEDURE COPYLST (CONSTS S : STRING;**
**VARS D : LSTRING);**

A string intrinsic procedure. Copies S to LSTRING D. The length of D is set to UPPER (S). An error occurs if the length of S is greater than the maximum length of D, i.e., if UPPER (S) > UPPER (D).

**PROCEDURE COPYSTR (CONSTS S : STRING; VARS D : STRING);**

A string intrinsic procedure. Copies S to STRING D. The remainder of D is set to blanks if UPPER (S) < UPPER (D). An error occurs if the length of S is greater than the maximum length of D, i.e., if UPPER (S) > UPPER (D).

**FUNCTION COS (X : NUMERIC) : REAL;**

An arithmetic function. Returns the cosine of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare CNSRQQ (CONSTS REAL4) and/or CNDRQQ (CONSTS REAL8) and use them instead.

**PROCEDURE DATE (VAR S : STRING);**

A clock procedure. If available, this procedure assigns the current date to its STRING (or LSTRING) variable. If an LSTRING is passed as the parameter, you must set the length you want before calling the procedure. The format depends on the target operating system.

**FUNCTION DECODE (CONST LSTR : LSTRING, X : M : N) : BOOLEAN;**

An extend level intrinsic function. Converts the character string in the LSTRING to its internal representation and assigns this to X. If the character string is not a valid external ASCII representation of a value whose type is assignment compatible with X, DECODE returns FALSE and the value of X is undefined.

DECODE works exactly the same as the READ procedure, including the use of M and N parameters (see Section 16.2.2, "READ Formats," for a discussion of these parameters). When X is a subrange, DECODE returns FALSE if the value is out of range (regardless of the setting of the range checking switch.) Leading and trailing spaces and tabs in the LSTRING are ignored. All other characters in the LSTRING must be part of the representation.

X must be one of the types INTEGER, WORD, enumerated, one of their subranges, BOOLEAN, REAL4, REAL8, INTEGER4, or a pointer (address types need the .R or .S suffix).

In a segmented memory environment, the LSTR parameter must reside in the default data segment.

See also FUNCTION ENCODE.

**PROCEDURE DELETE (VARS D : LSTRING; I, N : INTEGER);**

A string intrinsic procedure. Deletes N characters from D, starting with D [I]. An error occurs if an attempt is made to delete more characters starting at I than it is possible to delete, i.e., if D.LEN < (I + N - 1).

**PROCEDURE DISBIN;**

A library routine (interrupt). Along with ENABIN and VECTIN, DISBIN handles interrupt processing. DISBIN disables interrupts; ENABIN enables interrupts; VECTIN sets an interrupt vector. The effect of these procedures varies with the target machine.

**PROCEDURE DISCARD (VAR F);**

A file system procedure (extend level I/O). Closes and deletes an open file.

See Section 16.3.1, "Extend Level Procedures," for a description of DISCARD.

**PROCEDURE DISMQQ (BLOCK : ADSMEM);**

A library routine (segmented heap management function). This function takes an ADSMEM generated by ALLMQQ or GETMQQ and invokes FREMQQ to return the space described by the ADSMEM to the long heap memory pool. If errors occur, runtime messages are generated.

DISMQQ may not be available in your implementation of Microsoft Pascal.

**PROCEDURE DISPOSE (VARS P : POINTER);**

A dynamic allocation procedure (short form). Releases the memory used for the variable pointed to by P. P must be a valid pointer; it may not be NIL, uninitialized, or pointing at a heap item that already has been DISPOSEd. These are checked if the NIL check switch is on.

P should not be a reference parameter or a WITH statement record pointer, but these errors are not caught. A DISPOSE of a WITH statement record can be done at the end of the WITH statement without problem.

If the variable is a super array type or a record with variants, you may safely use the short form of DISPOSE to release the variable, regardless of whether it was allocated with the long or short form of NEW. Using the short form of DISPOSE on a heap variable allocated with the long form of NEW is an ISO-defined error not caught in MS-Pascal.

**PROCEDURE DISPOSE (VARS P : POINTER ; T1, T2, . . . TN : TAGS) ;**

A dynamic allocation procedure (long form). The long form of DISPOSE works the same as the short form. However, the long form checks the size of the variable against the size implied by the tag field or array upper bound values T1, T2, . . . Tn. These tag values should be the same as defined in the corresponding NEW procedure.

See also the SIZEOF function, which uses the same array upper bounds or tag value parameters to return the number of bytes in a variable.

**PROCEDURE ENABIN ;**

A library routine (interrupt handling). Along with DISBIN and VECTIN, ENABIN handles interrupt processing. ENABIN enables interrupts; DISBIN disables interrupts; VECTIN sets an interrupt vector. The effect of these procedures may vary with the target machine.

**FUNCTION ENCODE (VAR LSTR : LSTRING, X : M : N) : BOOLEAN ;**

An extend level intrinsic function. Converts the expression X to its external ASCII representation and puts this character string into LSTR. Returns TRUE, unless the LSTRING is too small to hold the string generated. In this case, ENCODE returns FALSE and the value of the LSTR is undefined. ENCODE works exactly the same as the WRITE procedure, including the use of M and N parameters (see Section 16.2.4, "WRITE Formats," for a discussion of these parameters).

X must be one of the types INTEGER, WORD, enumerated, one of their subranges, BOOLEAN, REAL4, REAL8, INTEGER4, or a pointer (address types need the .R or .S suffix).

In a segmented memory environment, the LSTR parameter must reside in the default data segment.

See also FUNCTION DECODE.

**PROCEDURE ENDOQQ;**

A library procedure (termination). ENDOQQ is called during termination and the default version does nothing. However, you may write your own version of ENDOQQ, if you want, to invoke a debugger or to write customized messages, such as the time of execution, to a terminal screen.

Since ENDOQQ is called after errors are processed, if ENDOQQ itself invokes an error, the result is an infinite termination loop.

See also PROCEDURE BEGOQQ.

**PROCEDURE ENDXQQ;**

The termination procedure. ENDXQQ is the overall termination routine and performs the following actions:

1. It calls ENDOQQ.
2. It terminates the file system (closing any open files).
3. It returns to the target operating system (or whatever called BEGXQQ).

ENDXQQ may be useful for ending program execution from inside a procedure or function, without calling ABORT. ENDXQQ corresponds to the HALT procedure in other Pascals.

**FUNCTION EOF : BOOLEAN;**
**FUNCTION EOF (VAR F) : BOOLEAN;**

A file system function. Indicates whether the current position of the file is at the end of the file F for SEQUENTIAL and TERMINAL file modes. EOF with no parameters is the same as EOF (INPUT).

See Section 16.1.3, "EOF and EOLN," for a more complete description of EOF.

## FUNCTION EOLN : BOOLEAN ;
## FUNCTION EOLN (VAR F) : BOOLEAN ;

A file system function. Indicates whether the current position of the file is at the end of a line in the textfile F. EOLN with no parameters is the same as EOLN (INPUT).

See Section 16.1.3, "EOF and EOLN," for a description of EOLN.

## PROCEDURE EVAL (EXPRESSION, EXPRESSION, . . . ) ;

An extend level intrinsic procedure. Evaluates expression parameters only, but accepts any number of parameters of any type. EVAL is used to evaluate an expression as a statement; it is commonly used to evaluate a function for its side effects only, without using the function return value.

## FUNCTION EXSRQQ (CONSTS A : REAL4) : REAL4 ;
## FUNCTION EXDRQQ (CONSTS A : REAL8) : REAL8 ;

See FUNCTION EXP.

## FUNCTION EXP (X : NUMERIC) : REAL ;

An arithmetic function. Returns the exponential value of X (i.e., e to the X). Both X and the return value are of type REAL. To force a particular precision, declare EXSRQQ (CONSTS REAL4) and/or EXDRQQ (CONSTS REAL8) and use them instead.

## PROCEDURE FILLC (D : ADRMEM ; N : WORD ; C : CHAR) ;

A system level intrinsic procedure. Fills D with N copies of the CHAR C. No bounds checking is done.

See also PROCEDURE FILLSC for segmented address types. The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

## PROCEDURE FILLSC (D : ADSMEM ; N : WORD ; C : CHAR) ;

A system level intrinsic procedure. Fills D with N copies of the CHAR C. No bounds checking is done.

See also PROCEDURE FILLC for relative address types. The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

## FUNCTION FLOAT (X : INTEGER) : REAL ;

A data conversion function. Converts an INTEGER value to a REAL value. You normally don't need this function, since INTEGER-to-REAL is usually done automatically. However, because FLOAT is needed by the runtime package, it is included at the standard level.

## FUNCTION FLOAT4 (X : INTEGER4) : REAL ;

A data conversion function. Converts an INTEGER4 value to a REAL value. This type conversion is also done automatically; however, it is possible to lose precision. (Losing precision is not an error.)

## FUNCTION FREECT (SIZE : WORD) : WORD ;

A library function. Returns an estimate of the number of times NEW could be called to allocate heap variables with length SIZE bytes. FREECT takes into account DISPOSE and adjacent free blocks and is generally used with the SIZEOF function. However, it does not assume any stack space will be needed. Since stack space generally will be needed, the value returned should be reduced accordingly.

Example:

```
IF FREECT (SIZEOF (REC, TRUE, 5)) > 2
  THEN DO_SOMETHING
```

**FUNCTION FREMQQ (BLOCK : ADSMEM) : WORD;**

A library routine (segmented heap management function). This function takes an ADSMEM generated by ALLMQQ or GETMQQ and returns the space described by the ADSMEM to the long heap memory pool. The function returns a word value. If the word value is 0, FREMQQ encountered no errors. If the word value is 1, the release of memory was in error. FREMQQ may not be available on your system.

**PROCEDURE GET (VAR F);**

A file system procedure. GET either reads the currently pointed-to component of F to the buffer variable F^ and advances the file pointer, or sets the buffer variable status to empty.

See Section 16.1.1, "GET and PUT," for a description of GET.

**FUNCTION GETMQQ (WANTS : WORD) : ADSMEM;**

A library routine (segmented heap management function). This function returns a long segmented address of type ADSMEM. GETMQQ takes a parameter "wants" that is a memory request in bytes.

GETMQQ invokes ALLMQQ but in addition, issues error messages if ALLMQQ fails. GETMQQ may not be available on your system.

**FUNCTION GTYUQQ (LEN : WORD; LOC : ADSMEM) : WORD;**

A library function (terminal I/O). Reads a maximum of LEN characters from the terminal keyboard and stores them in memory beginning at the address LOC. The return value is the number of characters actually read. GTYUQQ always reads the entire line you enter. Any characters typed beyond the end of the buffer length are lost.

Example:

```
    LSTR.LEN := GTYUQQ (UPPER(LSTR), ADS LSTR(1));
```

Together with PTYUQQ and PLYUQQ, GTYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the MS-Pascal file system.

### FUNCTION HIBYTE (INTEGER-WORD) : BYTE;

An extend level intrinsic function. Returns the most significant byte of an INTEGER or WORD. Depending on the target processor, the most significant byte may be the first or the second addressed byte of the word.

See also FUNCTION LOBYTE.

### FUNCTION HIWORD (INTEGER4) : WORD;

An extend level intrinsic function. Returns the high-order word of the four bytes of the INTEGER4. The sign bit of the INTEGER4 becomes the most significant bit of the WORD.

See also FUNCTION LOWORD.

### PROCEDURE INSERT (CONSTS S : STRING; VARS D : LSTRING; I : INTEGER);

A string intrinsic procedure. Inserts S starting just before D [I]. An error occurs if D is too small, i.e., if

```
UPPER (D) < UPPER (S) + D.LEN + 1
```

or if

```
D.LEN < I
```

### FUNCTION LADDOK (A, B : INTEGER4; VAR C : INTEGER4) : BOOLEAN;

A library routine (no-overflow arithmetic). Sets C equal to A plus B. One of two functions that do 32-bit signed arithmetic without causing a runtime error, even if the arithmetic debugging switch is on. Both LADDOK and LMULOK return TRUE if there is no overflow, and FALSE if there is. These routines are useful for extended-precision arithmetic, or modulo 2^32 arithmetic, or arithmetic based on user input data.

### FUNCTION LDSRQQ (CONSTS A : REAL4) : REAL4; FUNCTION LDDRQQ (CONSTS A : REAL8) : REAL8;

Arithmetic functions. Return the logarithm, base 10, of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**FUNCTION LMULOK**
**(A, B : INTEGER4 ; VAR C : INTEGER4) : BOOLEAN ;**

A library routine (no-overflow arithmetic). Sets C equal to A times B. One of two functions that do 32-bit signed arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. Both LMULOK and LADDOK return TRUE if there is no overflow, and FALSE if there is. These routines are useful for extended-precision arithmetic, or modulo 2^32 arithmetic, or arithmetic based on user input data.

**FUNCTION LN (X : REAL) : REAL ;**

An arithmetic function. Returns the logarithm, base e, of X. Both X and the return value are of type REAL. To force a particular precision, declare LNSRQQ (CONSTS REAL4) and/or LNDRQQ (CONSTS REAL8) and use them instead. An error occurs if X is less than or equal to zero.

**FUNCTION LNSRQQ (CONSTS A : REAL4) : REAL4 ;**
**FUNCTION LNDRQQ (CONSTS A : REAL8) : REAL8 ;**

See FUNCTION LN.

**FUNCTION LOBYTE (INTEGER-WORD) : BYTE ;**

An extend level intrinsic function. Returns the least significant byte of an INTEGER or WORD. Depending on the target processor, the least significant byte may be the first or the second addressed byte of the word.

See also FUNCTION HIBYTE.

**FUNCTION LOCKED (VARS SEMAPHORE : WORD) : BOOLEAN ;**

A library function (semaphore). If the semaphore was available, LOCKED returns the value TRUE and sets the semaphore unavailable. Otherwise, if it was already locked, LOCKED returns FALSE. UNLOCK sets the semaphore available. As a binary semaphore, there are only two states.

See also PROCEDURE UNLOCK.

**FUNCTION LOWER (EXPRESSION) : VALUE;**

An extend level intrinsic function. LOWER takes a single parameter of one of the following types: array, set, enumerated, or subrange. The value returned by LOWER is one of the following:

1. the lower bound of an array
2. the first allowable element of a set
3. the first value of an enumerated type
4. the lower bound of a subrange

LOWER uses the type, not the value, of the expression. The value returned by LOWER is always a constant.

See also FUNCTION UPPER.

**FUNCTION LOWORD (INTEGER4) : WORD;**

An extend level intrinsic function. Returns the low-order WORD of the four bytes of the INTEGER4.

See also FUNCTION HIWORD.

**PROCEDURE MARKAS (VAR HEAPMARK : INTEGER4);**

A library procedure (heap management). Parallels the MARK procedure in other Pascals. MARKAS marks the upper and lower limits of the heap. The DISPOSE procedure is generally more powerful, but MARKAS may be useful for converting from other Pascal dialects.

In other Pascals, the parameter is of a pointer type. However, MS-Pascal needs two words to save the heap limits, since in some implementations the heap grows toward both higher and lower addresses. The HEAPMARK variable should not be used as a normal INTEGER4 number; it should only be set by MARKAS and passed to RELEAS.

To use MARKAS and RELEAS, pass an INTEGER4 variable, M for example, as a VAR parameter to MARKAS. MARKAS places the bounds of the heap in M. To release heap space, simply invoke the procedure with RELEAS (M).

MARKAS and RELEAS work as intended only if you never call DISPOSE.

**FUNCTION MDSRQQ (CONSTS A, B : REAL4) : REAL4 ;**
**FUNCTION MDDRQQ (CONSTS A, B : REAL8) : REAL8 ;**

Arithmetic functions. A modulo B, defined as:

MDSRQQ (A, B) = A - AISRQQ (A/B) * B

MDDRQQ (A, B) = A - AIDRQQ (A/B) * B

Both A and B are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**FUNCTION MEMAVL : WORD ;**

A library function (heap management). Returns the number of bytes available between the stack and the heap. MEMAVL acts like the MEMAVAIL function in UCSD Pascal. If you have previously used DISPOSE, MEMAVL may return a value less than the actual number of bytes available.

**FUNCTION MNSRQQ (CONSTS A, B : REAL4) : REAL4 ;**
**FUNCTION MNDRQQ (CONSTS A, B : REAL8) : REAL8 ;**

Arithmetic functions. Return the value of A or B, whichever is smaller. Both A and B are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

See also FUNCTION MXSRQQ and FUNCTION MXDRQQ.

**PROCEDURE MOVEL (S, D : ADRMEM ; N : WORD);**

A system level intrinsic procedure. Moves N characters (bytes) starting at S^ to D^, beginning with the lowest addressed byte of each array. Regardless of the value of the range and index checking switches, there is no bounds checking.

Example:

```
MOVEL (ADR 'New String Value', ADR V, 16)
```

See also PROCEDURE MOVESL for segmented address types. Use MOVEL and MOVESL to shift bytes left or when the address ranges do not overlap.

The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

## PROCEDURE MOVER (S, D : ADRMEM ; N : WORD);

A system level intrinsic procedure. Like MOVEL, but starts at the highest addressed byte of each array. Use MOVER and MOVESR to shift bytes right. As with MOVEL, there is no bounds checking.

Example:

```
MOVER (ADR V[0], ADR V[4], 12)
```

See also PROCEDURE MOVESR for segmented address types.

The MOVEs and FILLs take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

## PROCEDURE MOVESL (S, D : ADSMEM ; : N : WORD);

A system level intrinsic procedure. Moves N characters (bytes) starting at S^ to D^, beginning with the lowest addressed byte of each array. Regardless of the value of the range and index checking switches, there is no bounds checking.

Example:

```
MOVESL (ADS 'New String Value', ADS V, 16)
```

See also PROCEDURE MOVEL for relative address types. Use MOVEL and MOVESL to shift bytes left or when the address ranges do not overlap.

The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

**PROCEDURE MOVESR (S, D : ADSMEM ; N : WORD) ;**

A system level intrinsic procedure. Like MOVESL, but starts at the highest addressed byte of each array. Use MOVER and MOVESR to shift bytes right. As with MOVESL, there is no bounds checking.

Example:

```
MOVESR (ADS V[0], ADS V[4], 12)
```

See also PROCEDURE MOVER for relative address types.

The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but since all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be used as the actual parameter. These are dangerous but sometimes useful procedures.

**FUNCTION MXSRQQ (CONSTS A, B : REAL4) : REAL4 ;**
**FUNCTION MXDRQQ (CONSTS A, B : REAL8) : REAL8 ;**

Arithmetic functions. Return the value of A or B, whichever is larger. Both A and B are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

See also FUNCTION MNSRQQ and MNDRQQ.

**PROCEDURE NEW (VARS P : POINTER) ;**

A library procedure (heap management, short form). Allocates a new variable V on the heap and at the same time assigns a pointer to V to the pointer variable P (a VARS parameter). The type of V is determined by the pointer declaration of P. If V is a super array type, use the long form of the procedure instead. If V is a record type with variants, the variants giving the largest possible size are assumed, permitting any variant to be assigned to P^.

## PROCEDURE NEW (VARS P : POINTER ; T1, T2, . . . TN : TAGS) ;

A library procedure (heap management, long form). Allocates a variable with the variant specified by the tag field values T1 through TN. The tag field values are listed in the order in which they are declared. Any trailing tag fields can be omitted.

If all tag field values are constant, MS-Pascal allocates only the amount of space required on the heap, rounded up to a word boundary. The value of any omitted tag fields is assumed to be such that the maximum possible size is allocated.

If some tag fields are not constant values, the compiler uses one of two strategies:

1. It assumes that the first nonconstant tag field and all following tags have unknown values, and allocates the maximum size necessary.
2. It generates a special runtime call to a function that calculates the record size from the variable tag values available. This depends on the implementation. A similar procedure applies to DISPOSE and SIZEOF.

You should set all tag fields to their proper values after the call to NEW and never change them. The compiler does not do any of the following:

1. assign tag values
2. check that they are initialized correctly
3. check that their value is not changed during execution

According to the ISO standard, a variable created with the long form of NEW cannot be any of the following:

1. used as an expression operand
2. passed as a parameter
3. assigned a value

MS-Pascal does not catch these errors. Fields within the record can be used normally.

Assigning a larger record to a smaller one allocated with the long form of NEW would wipe out part of the heap. This condition is difficult to detect at compile time. Therefore, in MS-Pascal, any assignment to a record in the heap that has variants uses the actual length of the record in the heap, rather than the maximum length.

However, an assignment to a field in an invalid variant may destroy part of another heap variable or the heap structure itself. This error is not caught, unless all tag values are explicit, the tag values are correct, and the tag checking switch is on.

The extend level allows pointers to super arrays. The long form of NEW is used as described above, except that array upper bound values are given instead of tag values. All upper bounds must be given. Bounds can be constants or expressions; in any case, only the size required is allocated.

The entire array referenced by such a pointer cannot be assigned or compared, except that LSTRINGs can always be compared. The entire array can be passed as a reference parameter if the formal parameter is of the same super array type. Components of the array can be used normally.

## FUNCTION ODD (X : ORDINAL) : BOOLEAN;

A data conversion function. Tests the ordinal value X to see whether it is odd. ODD is TRUE only if ORD (X) is odd; otherwise it is FALSE.

## FUNCTION ORD (X : VALUE) : INTEGER;

A data conversion function. Converts to INTEGER any value of one of the types shown in the following list according to the rules given.

| Type of X | Return value |
|---|---|
| INTEGER | X |
| WORD <= MAXINT | X |
| WORD > MAXINT | X - 2 * (MAXINT + 1) (i.e., same 16 bits as at start!) |
| CHAR | ASCII code for X |
| Enumerated | Position of X in the type definition, starting with 0 |
| INTEGER4 | Lower 16 bits (i.e., same as ORD (LOWORD (INTEGER4)) |
| Pointer | Integer value of pointer |

**PROCEDURE PACK (CONSTS A : UNPACKED; I : INDEX; VARS Z : PACKED);**

A data conversion procedure. Moves elements of an unpacked array to a packed array. If A is an ARRAY [M . . N] OF T and Z is a PACKED ARRAY [U . . V] OF T, then PACK (A, I, Z) is the same as:

```
FOR J := U TO V DO Z [J] := A [J - U + I]
```

In both PACK and UNPACK, the parameter I is the initial index within A. The bounds of the arrays and the value of I must be reasonable; i.e., the number of components in the unpacked array A from I to M must be at least as great as the number of components in the packed array Z. The range checking switch controls checking of the bounds.

**PROCEDURE PAGE;**
**PROCEDURE PAGE (VAR F);**

A file system procedure. Causes skipping to the top of a new page when the textfile F is printed. PAGE with no parameter is the same as PAGE (INPUT).

See Section 16.1.4, "PAGE," for a description of PAGE.

**FUNCTION PISRQQ**
**(CONSTS A : REAL4 ; CONSTS B : INTEGER4) : REAL4 ;**
**FUNCTION PIDRQQ**
**(CONSTS A : REAL8 ; CONSTS B : INTEGER4) : REAL8 ;**

Arithmetic functions. The return value is A**B (A to the INTEGER power of B). A is of type REAL4 or REAL8, as shown. B is always of type INTEGER4. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**PROCEDURE PLYUQQ ;**

A library routine (terminal I/O). Writes an end-of-line character to the terminal screen.

Together with GETYQQ and PTYUQQ, PLYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the MS-Pascal file system.

**FUNCTION POSITN**
**(CONSTS PAT : STRING ; CONSTS S : STRING ; I : INTEGER) : INTEGER ;**

A string intrinsic function. Returns the integer position of the pattern PAT in S, starting the search at S [I]. If PAT is not found or if I > upper (S), the return value is 0. If PAT is the null string, the return value is 1. There are no error conditions.

**FUNCTION PRED (X : ORDINAL) : ORDINAL ;**

A data conversion function. Determines the ordinal "predecessor" to X. The ORD of the result returned is equal to ORD (X) - 1. An error occurs if the predecessor is out of range or overflow occurs. These errors are caught if appropriate debug switches are on.

**FUNCTION PRSRQQ (A, B : REAL4) : REAL4;**
**FUNCTION PRDRQQ (A, B : REAL8) : REAL8;**

Arithmetic functions. The return value is A**B (A to the REAL power of B). Both A and B are of type REAL4 or REAL8, as shown. An error occurs if $A < 0$ (even if B happens to have an integer value). These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**PROCEDURE PTYUQQ (LEN : WORD; LOC : ADSMEM);**

A library routine (terminal I/O). Writes LEN characters, beginning at LOC in memory, to the terminal screen.

Example:

```
PTYUQQ (8, ADS 'PROMPT: ');
```

Together with GETYQQ and PLYUQQ, PTYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the MS-Pascal file system.

**PROCEDURE PUT (VAR F);**

A file system procedure. Writes the value of the file buffer variable F^ to the currently pointed-to component of F and advances the file pointer.

See Section 16.1.1, "GET and PUT," for a description of PUT.

**PROCEDURE READ (VAR F; P1, P2, . . . PN);**

A file system procedure. READ reads data from files. Both READ and READLN are defined in terms of the more primitive operation, GET.

See Section 16.2, "Textfile Input and Output," for a description of READ.

## PROCEDURE READFN (VAR F; P1, P2, . . . PN);

A file system procedure (extend level I/O). READFN is the same as READ (not READLN) with two exceptions:

1. File parameter F should be present (INPUT is assumed but a warning is given).
2. If a parameter P is of type FILE, a sequence of characters forming a valid filename is read from F and assigned to P in the same manner as ASSIGN.

Parameters of other types are read in the same way as the READ procedure.

See Section 16.3.1, "Extend Level Procedures," for a description of READFN.

## PROCEDURE READLN (VAR F; P1, P2, . . . PN);

A textfile I/O procedure. At the primitive GET level, without parameters, READLN (F) is equivalent to the following:

```
BEGIN
   WHILE NOT EOLN (F) DO GET (F);
   GET (F)
END
```

The procedure READLN is very much like READ, except that it reads up to and including the end of line.

See Section 16.2, "Textfile Input and Output," for a description of READ.

## PROCEDURE READSET (VAR F; VAR L : LSTRING; CONST S : SETOFCHAR);

A file system procedure (extend level I/O). READSET reads characters and puts them into L, as long as the characters are in the set S and there is room in L.

See Section 16.3.1, "Extend Level Procedures," for a description of READSET.

### PROCEDURE RELEAS (VAR HEAPMARK : INTEGER4);

A library routine (heap management). Parallels the RELEASE procedure in other Pascals. RELEAS disposes of heap space past the area set with a previous MARKAS call. The DISPOSE procedure in MS-Pascal is generally more powerful, but RELEAS may be useful for converting from other Pascal dialects.

In other Pascals, the parameter is of a pointer type. However, MS-Pascal needs two words to save the heap limits, since in some implementations the heap grows toward both higher and lower addresses. The HEAPMARK variable should not be used as a normal INTEGER4 number; it should only be set by MARKAS and passed to RELEAS.

To use MARKAS and RELEAS, pass an INTEGER4 variable, M for example, as a VAR parameter to MARKAS. MARKAS places the bounds of the heap in M. To RELEAS heap space, simply invoke the procedure with RELEAS (M).

MARKAS and RELEAS work as intended only if DISPOSE is never called.

### PROCEDURE RESET (VAR F);

A file system procedure. Resets the current file position to its beginning and does a GET (F).

See Section 16.1.2, "RESET and REWRITE," for a description of RESET.

### FUNCTION RESULT (FUNCTION-IDENTIFIER) : VALUE;

An extend level intrinsic function. Used to access the current value of a function; can only be used within the body of the function itself or in a procedure or function nested within it.

### FUNCTION RETYPE (TYPE-IDENT, EXPRESSION) : TYPE-IDENT;

A system level intrinsic function. Provides a generic type escape, returns the value of the given expression as if it had the type named by the type identifier. The types implied by the type identifier and the expression should usually have the same length, but this is not required. RETYPE for a structure can be followed by component selectors (array index, fields, reference, etc.). RETYPE is a "dangerous" type escape and may not work as intended.

Example:

```
TYPE COLOR = (RED, BLUE, GREEN);
       S2 = STRING (2) ;
VAR C :CHAR;
      I, J :INTEGER;
      R :REAL4;
      TINT :COLOR;
      .
      .
      R := RETYPE (REAL4, 'abcd');
      {Here, a 4-byte string literal is}
      {converted into a real number.}
      {Note that REAL4 numbers also}
      {require 4 bytes.}

      TINT := RETYPE (COLOR, 2)
      {Here, 2 is converted into a color,}
      {which in this case is GREEN.}
      {This is a relatively "safe" use}
      {of the RETYPE function.}

      C := RETYPE (S2, I) [J]
      {Here, I is retyped into a two}
      {character string. Then J selects}
      {a single character of the string}
      {which is assigned to C.}
```

There are two other ways to change type in MS-Pascal.

1. First, you can declare a record with one variant of each type needed, assign an expression to one variant, and then get the value back from another variant. (This is an error not caught at the standard level. Note that the relative mapping of variables is subject to change between different versions of the compiler.)

2. Second, you can declare an address variable of the type wanted and assign to it the address of any other variable (using ADR).

Each of these methods has its own subtle differences and quirks and should be avoided whenever possible.

## PROCEDURE REWRITE (F);

A file system procedure. Resets the current file position to its beginning.

See Section 16.1.2, "RESET and REWRITE," for a description of REWRITE.

## FUNCTION ROUND (X : REAL) : INTEGER;

An arithmetic function. Rounds X away from zero. X is of type REAL4 or REAL8; the return value is of type INTEGER. The effect of ROUND on a number with a fractional part of 0.5 varies with the implementation.

Examples:

```
ROUND (1.6) is 2
ROUND (-1.6) is -2
```

An error occurs if ABS (X + 0.5) >= MAXINT.

## FUNCTION ROUND4 (X : REAL) : INTEGER4;

An arithmetic function. Rounds real X away from zero. X is of type REAL4 or REAL8; the return value is of type INTEGER4. The effect of ROUND4 on a number with a fractional part of 0.5 varies with the implementation.

Examples:

```
ROUND4 (1.6) is 2
ROUND4 (-1.6) is -2
```

An error occurs if ABS (X + 0.5) >= MAXINT4.

## FUNCTION SADDOK (A, B : INTEGER; VAR C : INTEGER) : BOOLEAN;

A library routine (no-overflow arithmetic). Sets C equal to A plus B. One of two functions that do 16-bit signed arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. Both SADDOK and SMULOK return TRUE if there is no overflow, and FALSE if there is. These routines can be useful for extended-precision arithmetic, or modulo 2^16 arithmetic, or arithmetic based on user input data.

**FUNCTION SCANEQ (LEN : INTEGER; PAT : CHAR; CONSTS S : STRING; I : INTEGER) : INTEGER;**

A string intrinsic function. Scans, starting at S [I], and returns the number of characters skipped. SCANEQ stops scanning when a character equal to pattern PAT is found or LEN characters have been skipped. If LEN < 0, SCANEQ scans backwards and returns a negative number. SCANEQ returns the LEN parameter if it finds no characters equal to pattern PAT found or if I > UPPER (S). There are no error conditions.

**FUNCTION SCANNE (LEN : INTEGER; PAT : CHAR; CONSTS S : STRING; I : INTEGER) : INTEGER;**

A string intrinsic function. Like SCANEQ, but stops scanning when a character not equal to pattern PAT is found.

Scans, starting at S [I], and returns the number of characters skipped. SCANNE stops scanning when a character not equal to pattern PAT is found or LEN characters have been skipped. If LEN < 0, SCANNE scans backwards and returns a negative number. SCANNE returns LEN parameter if it finds all characters equal to pattern PAT found or if I > UPPER (S). There are no error conditions.

**PROCEDURE SEEK (VAR F; N : INTEGER4);**

A file system procedure (extend level I/O). In contrast to normal sequential files, DIRECT files are random access structures. SEEK is used to randomly access components of such files.

See Section 16.3, "Extend Level I/O," for details.

**FUNCTION SHSRQQ (CONSTS A : REAL4) : REAL4;**
**FUNCTION SHDRQQ (CONSTS A : REAL8) : REAL8;**

Arithmetic functions. Return the hyperbolic sine of A. A is of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**FUNCTION SIN (X : NUMERIC) : REAL ;**

An arithmetic function. Returns the sine of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare SNSRQQ (CONSTS REAL4) and/or SNDRQQ (CONSTS REAL8) and use them instead.

**FUNCTION SIZEOF (VARIABLE) : WORD ;**
**FUNCTION SIZEOF (VARIABLE, TAG1, TAG2, . . . TAGN) : WORD ;**

An extend level intrinsic function. Returns the size of a variable in bytes. Tag values or array upper bounds are set as in the NEW and DISPOSE functions. If the variable is a record with variants, and the first form is used, the maximum size possible is returned. If the variable is a super array, the second form, which gives upper bounds, must be used.

**FUNCTION SMULOK (A, B : INTEGER ; VAR C : INTEGER) : BOOLEAN ;**

A library routine (no-overflow arithmetic function). Sets C equal to A times B. One of two functions that do 16-bit signed arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error, even if the arithmetic debugging switch is off. Each routine returns TRUE if there is no overflow, and FALSE if there is. These routines can be useful for extended-precision arithmetic, or modulo 2^16 arithmetic, or arithmetic based on user input data.

**FUNCTION SNSRQQ (CONSTS A : REAL4) : REAL4 ;**
**FUNCTION SNDRQQ (CONSTS A : REAL8) : REAL8 ;**

See FUNCTION SIN.

**FUNCTION SQR (X : NUMERIC) : NUMERIC ;**

An arithmetic function. Returns the square of X, where X is of type REAL, INTEGER, WORD, or INTEGER4.

**FUNCTION SQRT (X) : REAL**

An arithmetic function. Returns the square root of X, where X is of type REAL. To force a particular precision, declare SRSRQQ (CONSTS REAL4) and/or SRDRQQ (CONSTS REAL8) and use them instead. An error occurs if X is less than 0.

**FUNCTION SRSRQQ (CONSTS A : REAL4) : REAL4 ;**
**FUNCTION SRDRQQ (CONSTS A : REAL8) : REAL8 ;**

See FUNCTION SQRT.

**FUNCTION SUCC (X : ORDINAL) : ORDINAL ;**

A data conversion function. Determines the ordinal "successor" to X. The ORD of the returned result is equal to ORD (X) + 1. An error occurs if the successor is out of range or overflow occurs. These errors are caught if appropriate debug switches are on.

**FUNCTION THSRQQ (CONSTS A : REAL4) : REAL4 ;**
**FUNCTION THDRQQ (CONSTS A : REAL8) : REAL8 ;**

Arithmetic functions. Return the hyperbolic tangent of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

**FUNCTION TICS : WORD ;**

A library routine (clock function). If available, TICS returns the value of an operating system timing location. The result is in a time interval, such as hundredths of a second, depending on the target operating system.

**PROCEDURE TIME (VAR S : STRING) ;**

A library routine (clock function). If available, this procedure assigns the current time to its STRING (or LSTRING) variable. If the parameter is an LSTRING, you must set the length before you call the TIME procedure. The format depends on the target operating system.

See also PROCEDURE DATE.

**FUNCTION TNSRQQ (CONSTS A : REAL4) : REAL4 ;**
**FUNCTION TNDRQQ (CONSTS A : REAL8) : REAL8 ;**

Arithmetic functions. Return the tangent of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the MS-FORTRAN runtime library and must be declared EXTERN before use.

## FUNCTION TRUNC (X : REAL) : INTEGER;

An arithmetic function. Truncates X toward zero. X is of type REAL4 or REAL8, and the return value is of type INTEGER.

Examples:

```
TRUNC (1.6) is 1
TRUNC (-1.6) is -1
```

An error occurs if ABS (X - 1.0) >= MAXINT.

## FUNCTION TRUNC4 (X : REAL) : INTEGER4;

An arithmetic function. Truncates real X towards zero. X is of type REAL4 or REAL8, and the return value is of type INTEGER4.

Examples:

```
TRUNC4 (1.6) is 1
TRUNC4 (-1.6) is -1
```

An error occurs if ABS (X - 1.0) >= MAXINT4.

## FUNCTION UADDOK (A, B : WORD; VAR C : WORD) : BOOLEAN;

A library routine (no-overflow arithmetic function). Sets C equal to A plus B. One of two functions that do 16-bit unsigned arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. The following is the binary carry resulting from this addition of A and B:

```
WRD (NOT UADDOK (A, B, C))
```

Both UADDOK and UMULOK return TRUE if there is no overflow and FALSE if there is. These routines are useful for extended-precision arithmetic, or modulo 2^16 arithmetic, or arithmetic based on user input data.

## FUNCTION UMULOK (A, B : WORD; VAR C : WORD) : BOOLEAN;

A library routine (no-overflow arithmetic function). Sets C equal to A times B. One of two functions that do 16-bit unsigned arithmetic without causing a runtime error on overflow. Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. Each routine returns TRUE if there is no overflow and FALSE if there is. These routines are useful for extended-precision arithmetic, or modulo 2^16 arithmetic, or arithmetic based on user input data.

## PROCEDURE UNLOCK (VARS SEMAPHORE : WORD);

A library routine (semaphore procedure). UNLOCK sets the semaphore available. As a binary semaphore, there are only two states. UNLOCK can be called any number of times and can be used to initialize the semaphore.

See also FUNCTION LOCKED.

## PROCEDURE UNPACK (CONSTS Z : PACKED; VARS A : UNPACKED; I : INDEX);

A data conversion procedure. Moves elements from packed array to an unpacked array. If A is an ARRAY [M . . N] OF T, and Z is a PACKED ARRAY [U . . V] OF T then the above call is the same as:

```
FOR J := U TO V DO A [J - U + I] := Z [J]
```

In both PACK and UNPACK, the parameter I is the initial index within A. The bounds of the arrays and the value of I must be reasonable; i.e., the number of components in the unpacked array A from I to M must be at least as great as the number of components in the packed array Z. The range checking switch controls checking of the bounds.

See also PROCEDURE PACK.

## FUNCTION UPPER (EXPRESSION) : VALUE;

An extend level intrinsic function. UPPER, like LOWER, takes a single parameter of one of the following types: array, set, enumerated, or subrange. The value returned by UPPER is one of the following:

1. the upper bound of an array
2. the last allowable element of a set

3. the last value of an enumerated type
4. the upper bound of a subrange

The value returned by UPPER is always a constant, unless the expression is of a super array type. In this case, the actual upper bound of the super array type is returned. Note that the type and not the value of the expression is used for UPPER.

See also FUNCTION LOWER.

**PROCEDURE VECTIN (V : WORD; PROCEDURE I [INTERRUPT]);**

A library routine (interrupt handling procedure). One of three procedures for processing interrupts. VECTIN sets an interrupt vector, so that interrupts of type V are connected to procedure I. (ENABIN enables interrupts and DISBIN disables interrupts.) The effect of these procedures and the meaning of V varies with the target machine.

**FUNCTION WRD (X : VALUE) : WORD;**

A data conversion procedure. Converts to WORD any of the types shown in the following list according to the rules given.

| **Type of X** | **Return Value** |
|---|---|
| WORD | X |
| INTEGER >= 0 | X |
| INTEGER < 0 | X + MAXWORD + 1 (i.e., same 16 bits as at start!) |
| CHAR | ASCII code for X |
| Enumerated | Position of X in the type definition, starting with 0 |
| INTEGER4 | Lower 16 bits (i.e., same as LOWORD(INTEGER4)) |
| Pointer | Word value of pointer |

**PROCEDURE WRITE (VAR F; P1, P2, . . . PN);**
**PROCEDURE WRITELN (VAR F; P1, P2, . . . PN);**

File system level intrinsic procedures. WRITES data to files. WRITE and WRITELN are defined in terms of the more primitive operation PUT. WRITELN is the same as WRITE, except it also writes an end-of-line.

See Section 16.2.3, "WRITE and WRITELN," for descriptions of these procedures.

# Chapter 16
# File-Oriented Procedures and Functions

Chapter 14, "Introduction to Procedures and Functions," introduced you to procedures and functions in general and described their use and construction.

Chapter 15, "Available Procedures and Functions," described eight categories of procedures and functions that are available to you either because they are predeclared or because they are part of the Microsoft Pascal runtime libraries. All except those that relate to file input and output were discussed in detail.

This chapter, Chapter 16, "File-Oriented Procedures and Functions," discusses all of the file I/O procedures and functions, as well as lazy evaluation and concurrent I/O, two special MS-Pascal features that facilitate your use of files.

The MS-Pascal file system supports a variety of procedures and functions that operate on files of different modes and structures. These procedures and functions can be categorized as shown in Table 16.1.

**Table 16.1**

**File System Procedures and Functions**

| Category | Procedures | Functions |
|---|---|---|
| Primitive | GET<br>PAGE<br>PUT<br>RESET<br>REWRITE | EOF<br>EOLN |
| Textfile I/O | READ<br>READLN<br>WRITE<br>WRITELN | |
| Extend Level I/O | ASSIGN<br>CLOSE<br>DISCARD<br>READSET<br>READFN<br>SEEK | |

## 16.1 File System Primitive Procedures and Functions

This section describes the seven primitive file system procedures and functions, which perform file I/O at the most basic level. Later, descriptions of READ and WRITE procedures are defined in terms of the primitives GET and PUT. Two related topics are also discussed in this section: lazy evaluation and concurrent I/O. In all descriptions which follow, F is a file parameter (files are always reference parameters), and F^ is the buffer variable.

In a segmented environment, all file variables operated on by these procedures must reside in the default data segment. This restriction increases the efficiency of file system calls.

1. GET and PUT

   The primitive procedures GET and PUT read to and write from the buffer variable F^. GET assigns the next component of a file to the buffer variable. PUT performs the inverse operation and writes the value of the buffer variable to the next component of the file F.

2. RESET and REWRITE

   The procedures RESET and REWRITE set the current position of a file to its beginning. RESET prepares for later GET and READ procedures. REWRITE prepares for later PUT and WRITE procedures.

3. EOF and EOLN

   The functions EOF and EOLN are used to check for the end-of-file and end-of-line conditions. They return a BOOLEAN result. In general, these values indicate when to stop reading a line or a file.

4. PAGE

   The procedure PAGE helps in formatting textfiles. It is not a necessary procedure in the same sense as GET and PUT.

## 16.1.1 GET and PUT

The primitive procedures GET and PUT are used to read to and write from the buffer variable, F^. GET assigns the next component of a file to the buffer variable. PUT performs the inverse operation and writes the value of the buffer variable to the next component of the file F.

### PROCEDURE GET (VAR F);

A primitive file system intrinsic procedure. If there is a next component in the file F, then:

1. The current file position is advanced to the next component.
2. The value of this component is assigned to the buffer variable F^.
3. EOF (F) becomes FALSE.

Advancing and assigning may be deferred internally, depending on the mode of the file.

If no next component exists, then EOF (F) becomes TRUE and the value of F^ becomes undefined. EOF (F) must be FALSE before GET (F), since reading past the end of file produces a runtime error. However, if F has mode DIRECT, EOF (F) can be TRUE or FALSE, since DIRECT mode permits repeated GET operations at the end of the file. If F^ is a record with variants, the compiler reads the variant with the maximum size.

### PROCEDURE PUT (VAR F);

A primitive file system intrinsic procedure. Writes the value of the file buffer variable F^ at the current file position and then advances the position to the next component.

1. For SEQUENTIAL and TERMINAL mode files, PUT is permitted if the previous operation on F was a REWRITE, PUT, or other WRITE procedure, and if it was not a RESET, GET, or other READ procedure.
2. For DIRECT mode files, PUT may occur immediately after a RESET or GET.

Exceptions to these rules cause errors to be generated. The value of F^ always becomes undefined after a PUT.

In MS-Pascal, the value of F^ after a PUT (F) may vary, depending on the target operating system and type of file. EOF (F) must be TRUE before PUT (F), unless F is a DIRECT mode file. EOF (F) is always TRUE after PUT (F). If F^ is a record with variants, the variant with the maximum size is written.

## 16.1.2 RESET and REWRITE

The procedures RESET and REWRITE are used to set the current position of a file to its beginning. RESET is used to prepare for later GET and READ operations. REWRITE is used to prepare for later PUT and WRITE operations.

**PROCEDURE RESET (VAR F);**

A primitive file system intrinsic procedure. Resets the current file position to its beginning and does a GET (F). If the file is not empty, the first component of F is assigned to the buffer variable F^, and EOF (F) becomes false. If the file is empty, the value of F^ is undefined and EOF (F) becomes true. RESET initializes a file F prior to its being read. For DIRECT files, writing can be done after RESET as well.

In MS-Pascal, a RESET closes the file and then opens it in a way that is dependent on the operating system. An error occurs if the filename has not been set (as a program parameter or with ASSIGN or READFN) or if the file cannot be found by the operating system. If an error occurs during RESET, the file is closed, even if the file was opened correctly and the error came with the initial GET.

RESET (INPUT) is done automatically when a program is initialized, but is also allowed explicitly. RESET on a file with mode DIRECT allows either reading or writing, but the file is not created automatically. Also, the initial GET reads record number one on a DIRECT mode file.

Note that an explicit GET (F) immediately following a RESET (F) assigns the second component of the file to the buffer variable. However, a READ (F, X) following a RESET (F) sets X to the first component of F, since READ (F, X) is "X := F^; GET (F)".

**PROCEDURE REWRITE (VAR F);**

A primitive file system intrinsic procedure. Positions the current file to its beginning. The value of F^ is undefined and EOF (F) becomes TRUE. This is needed to initialize a file F before writing (for DIRECT files, reading can be done after REWRITE too).

REWRITE closes and reopens a file. Before using REWRITE, the filename must be set, either as a program parameter, or with ASSIGN or READFN. If the file already exists and has mode SEQUENTIAL, the old contents are destroyed. If the file has mode DIRECT, the procedure returns an error; the existing contents are saved. Use RESET to update an existing DIRECT mode file. If an error occurs during REWRITE, the file is closed. If possible, an existing file with the same name is not affected when a REWRITE error occurs, but with some target operating systems the existing file may be deleted.

REWRITE (OUTPUT) is done automatically when a program is initialized, but can also be done explicitly if desired. REWRITE on a DIRECT mode file allows both reading and writing. REWRITE does not do an initial PUT the way RESET does an initial GET.

## 16.1.3 EOF and EOLN

The functions EOF and EOLN check for end-of-file and end-of-line conditions, respectively. They return a BOOLEAN result. In general, these values indicate when to stop reading a line or a file.

**FUNCTION EOF : BOOLEAN;**
**FUNCTION EOF (VAR F) : BOOLEAN;**

A primitive file system intrinsic function. Indicates whether the buffer variable F^ is positioned at the end of the file F for SEQUENTIAL and TERMINAL file modes. Therefore, if EOF (F) is TRUE, either the file is being written or the last GET has reached the end of the file.

With the DIRECT file mode, if EOF (F) is TRUE, either the last operation was a WRITE (the file may or may not be positioned at the end in this case) or the last GET reached the end of the file.

EOF without a parameter is equivalent to EOF (INPUT). EOF (INPUT) is generally never TRUE, except in some operating systems where a particular terminal character generates an end-of-file status, or if INPUT is reassigned to another file. Calling the EOF (F) function accesses the buffer variable F^.

**FUNCTION EOLN : BOOLEAN;**
**FUNCTION EOLN (VAR F) : BOOLEAN;**

A primitive file system intrinsic function. Indicates whether the current position of the file is at the end of a line in the textfile F after a GET (F). The file must have ASCII structure.

According to the ISO standard, calling EOLN (F) when EOF (F) is TRUE is an error. In MS-Pascal, this error is caught in most cases. The file F must be a file of type TEXT.

If EOLN (F) is TRUE, the value of F^ is a space, but the file is positioned at a line marker. EOLN without a parameter is equivalent to EOLN (INPUT). Calling the EOLN (F) function accesses the buffer variable F^.

## 16.1.4 PAGE

The procedure PAGE helps in formatting textfiles. It is not a "necessary" procedure in the same sense as GET and PUT.

**PROCEDURE PAGE;**
**PROCEDURE PAGE (VAR F);**

A primitive file system intrinsic procedure. Causes skipping to the top of a new page when the textfile F is printed. Since PAGE writes to the file, the initial conditions described for PUT must be TRUE. The file must have ASCII structure. PAGE without a parameter is equivalent to PAGE (OUTPUT).

If F is not positioned at the start of a line, PAGE (F) first writes a line marker to F. If F has mode SEQUENTIAL or DIRECT, then PAGE (F) writes a form feed, CHR (12). If F has mode TERMINAL, the effect is defined by the target operating system interface, which will usually also write a form feed.

## 16.1.5 Lazy Evaluation

Lazy evaluation is designed to solve a recurring problem in Pascal, specifically, how to READ from a terminal in a natural way.

The underlying problem is that the ISO standard defines the procedure RESET with an initial GET. Although acceptable in Pascal's original batch processing, sequential file environment, this kind of read-ahead doesn't work for interactive I/O.

Lazy evaluation in MS-Pascal provides for deferring actual physical input (textfiles only) when a buffer variable is evaluated.

For example, if a normal file is RESET and then READ, the RESET procedure calls the GET procedure, which sets the buffer variable to the first component of the file. However, if the file is a terminal, this first component does not yet exist!

Therefore, at a terminal, you must first type a character to accommodate the GET procedure. Only then would you be prompted for any input.

Lazy evaluation eliminates this problem for textfiles by giving the file's buffer variable a special status value that is either "full" or "empty."

The normal condition after a GET (F) is empty. The status is full after a buffer variable has been assigned to or assigned from. Full implies that the buffer variable value is equal to the currently pointed-to component. Empty implies just the opposite, that the buffer variable value does not equal the value of the currently pointed-to component and input to the buffer variable has been deferred. Table 16.2 summarizes these rules.

**Table 16.2**

**Lazy Evaluation**

| Statement | Status at call | Action | Status on exit |
|---|---|---|---|
| GET (F) | Full | Point to next file component. Becomes EMPTY since value pointed to is not in buffer variable. | Empty |
| GET (F) | Empty | Load buffer variable with current file component, then point to next file component. Becomes EMPTY since value pointed to is not in buffer variable. | Empty |
| Reference to F^ | Full | No action required. | Full |
| Reference to F^ | Empty | Load buffer variable with current file component. | Full |

Note that RESET (F) first sets the status full and then calls GET, which sets the status to empty without any physical input.

Example of lazy evaluation with automatic REWRITE call:

```
{INPUT is automatically a textfile.}
{RESET (INPUT); done automatically.}
WRITE (OUTPUT, 'Enter number: ');
READLN (INPUT, FOO);
```

The automatic initial call to the RESET procedure calls a GET procedure, which changes the buffer variable status from full to empty. The first physical action to the terminal is the prompt output from the WRITE. READLN does a series of the following operations:

```
temp := INPUT^;
GET (INPUT)
```

Physical input occurs when each INPUT^ is fetched and the GET procedure sets the status back to empty.

READLN ends with the sequence:

```
WHILE NOT EOLN DO GET (INPUT);
GET (INPUT)
```

This operation skips trailing characters and the line marker. The EOLN function invokes the physical input. Entering the carriage return sets the EOLN status. Both the GET procedure in the WHILE loop and the trailing GET set the status back to empty. The last physical input in the sequence above is reading the carriage return.

## 16.1.6 Concurrent I/O

On operating systems that support it, concurrent I/O permits a GET or PUT procedure to initiate the I/O and immediately return to the calling program. It is only used for BINARY structure files.

The program can do computation while the buffer variable is being filled or emptied. The buffer variable has another special status value that can be "ready" or "busy." If the status is busy when the buffer variable needs to be accessed, the program must wait until the status becomes ready.

For example, the following program fragment reads the file IN_FILE, does some computation with the current value, and then writes it to the file OUT_FILE:

```
WHILE NOT EOF (IN_FILE) DO
    {Check for end of input and}
    {wait until IN_ FILE^ is ready.}

BEGIN
    READ (IN_FILE, BUFF);
    {IN_ FILE^ is ready, so assign it to BUFF;}
    {start reading next component.}

    OPERATE (BUFF);
    {Go process value during READ and WRITE.}

    WRITE (OUT_FILE, BUFF)
    {Wait until OUT_ FILE^ is ready,}
    {then assign BUFF and start writing.}

END
```

The preceding example uses READ and WRITE procedures. Note that the following two lines are equivalent:

```
    READ (IN_FILE, BUFF)
    BUFF := IN_FILE^; GET (IN_FILE)
```

So are these two:

```
    WRITE (OUT_FILE, BUFF)
    OUT_FILE^ := BUFF; PUT (OUT_FILE)
```

Concurrent I/O applies to the procedures GET and PUT, as well as to the procedures READ and WRITE. In practice, it is unusual for the Microsoft Pascal runtime system to handle concurrency.

When accessing the buffer variable, either for lazy evaluation or concurrency, MS-Pascal generates an I/O system call. However, if the buffer variable is an actual reference parameter, the procedure or function using that parameter can do I/O to the same file, and these special calls cannot be executed.

Passing any buffer variable as a reference parameter is an error in MS-Pascal, although only a warning is given. Calling GET or PUT has an undefined effect on a file buffer variable accessed indirectly through a reference parameter. Assigning the address of a buffer variable to an address type variable is equally dangerous, since this bypasses the lazy evaluation and concurrency mechanisms.

## 16.2 Textfile Input and Output

Human-readable input and output in standard Pascal are done with textfiles. Textfiles are files of type TEXT and always have ASCII structure. Normally, the standard textfiles INPUT and OUTPUT are given as program parameters in the PROGRAM heading:

```
    PROGRAM IN_AND_OUT (INPUT, OUTPUT);
```

Other textfiles usually represent some input or output device such as a terminal, a card reader, a line printer, or an operating system disk file. The extend level permits using additional files not given as program parameters.

In order to facilitate the handling of textfiles, the four standard procedures READ, READLN, WRITE, and WRITELN are provided in addition to the procedures GET and PUT.

1. READ and READLN

   The procedures READ and READLN read data from textfiles. READ and READLN are defined in terms of the more primitive operation, GET. The procedure READLN is very much like READ, except that it reads up to and including the end of line.

2. WRITE and WRITELN

   The procedures WRITE and WRITELN write data to textfiles. WRITE and WRITELN are defined in terms of the more primitive operation, PUT. The procedure WRITELN writes a line marker to the end of a line. In all other respects, WRITELN is analogous to WRITE.

These procedures are more flexible in the syntax for their parameter lists, allowing, among other things, for a variable number of parameters. Moreover, the parameters need not necessarily be of type CHAR, but can also be of certain other types, in which case the data transfer is accompanied by an implicit data conversion operation. In some cases, parameters can include additional formatting values that affect the data conversions used.

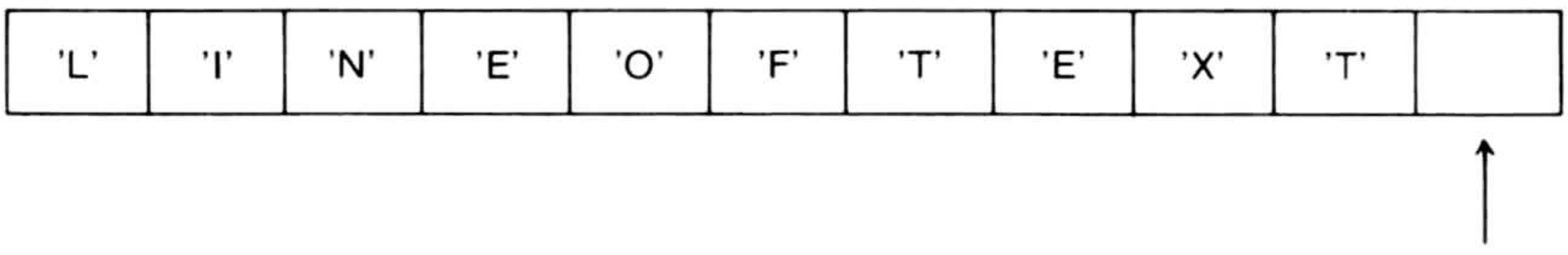
If the first variable is a file variable, then it is the file to be read or written. Otherwise, the standard files INPUT and OUTPUT are automatically assumed as default values in the cases of reading and writing, respectively.

These two files have TERMINAL mode and ASCII structure and are predeclared as:

```
    VAR INPUT, OUTPUT : TEXT;
```

In MS-Pascal, the files INPUT and OUTPUT are treated like other textfiles. They can be used with ASSIGN, CLOSE, RESET, REWRITE, and the other procedures and functions. However, even if present as program parameters, they are not initialized with a filename. Instead, they are assigned to the user's terminal. RESET of INPUT and REWRITE of OUTPUT are done automatically, whether or not they are present as program parameters.

Textfiles represent a special case among file types insofar as they are structured into lines by "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) yields the value true. For example:



{EOLN = TRUE} {F^ = ' '}

Advancing the file position once more causes one of three things to happen:

1. If the end of the file is reached, then EOF (F) becomes TRUE.
2. If the next line is empty, a blank is assigned to F^ and EOLN (F) remains TRUE.
3. Otherwise, the first character of the next line is assigned to F^ and EOLN (F) is set to FALSE.

Since line markers are not elements of type CHAR in standard Pascal, they can, in theory, only be generated by the procedure WRITELN. However, in MS-Pascal, an actual character may be used for the line marker. It may, therefore, be possible to WRITE a line marker, but not to READ one.

When a textfile being written is closed, a final line marker is automatically appended to the last line of any nonempty file in which the last character is not already a line marker.

When a textfile being read reaches the end of a nonempty file, a line marker for the last line is returned even if one was not present in the file. Therefore, lines in a textfile always end with a line marker.

Any list of data written by a WRITELN is usually readable with the same list in a READLN (unless an LSTRING occurs that is not on the end of the list).

Interactive prompt and response is very easy in MS-Pascal. To have input on the same line as the response, use WRITE for the prompt. READLN must always be used for the response. For example:

```
WRITE ('Enter command: ');
READLN (response);
```

If no file is given, most of the textfile procedures and functions assume either the INPUT file or the OUTPUT file. For example, if I is of type INTEGER, then READ (I) is the same as READ (INPUT, I).

## 16.2.1 READ and READLN

### PROCEDURE READ
### PROCEDURE READLN

File system intrinsic procedures for textfile I/O. READ and READLN read data from textfiles. Both are defined in terms of the more primitive operation, GET. That is, if P is of type CHAR, then READ (F, P) is equivalent to:

```
BEGIN
    P := F^;
    {Assign buffer variable F^ to P.}
    GET (F)
    {Assign next component of file to F^.}
END
```

READ can take more than a single parameter, as in READ (F, P1, P2, . . . Pn). This is equivalent to the following:

```
BEGIN
     READ (F, P1);
     READ (F, P2);
          .
          .
     READ (F, Pn)
END
```

The procedure READLN is very much like READ, except that it reads up to and including the end-of-line. At the primitive GET level, without parameters, READLN is equivalent to the following:

```
BEGIN
   WHILE NOT EOLN (F) DO GET (F);
   GET (F)
END
```

A READLN with parameters, as in READLN (F, P1, P2, . . . Pn), is equivalent to the following:

```
BEGIN
   READ (F, P1, P2, Pn);
   READLN (F)
END
```

READLN is often used to skip to the beginning of the next line. It can only be used with textfiles (ASCII mode).

If no other file is specified, both READ and READLN read from the standard INPUT file. Therefore, the name INPUT need not be designated explicitly. For example, these two READ statements perform identical actions:

```
READ (P1, P2, P3)
{Reads INPUT by default}
READ (INPUT, P1, P2, P3)
```

At the standard level, parameters P1, P2, and P3 above must be of one of the following types:

CHAR

INTEGER

REAL

The extend level also allows READ variables of the following types:

WORD
an enumerated type
BOOLEAN
INTEGER4
a pointer type
STRING
LSTRING

When the compiler reads a variable of a subrange type, the value read must be in range. Otherwise, an error occurs, regardless of the setting of the range checking switch.

The procedure READ can also read from a file that is not a textfile (e.g., has BINARY mode). The form READ (F, P1, P2, . . . Pn) can be used on a BINARY file. However, this READ will not work as expected after a SEEK on a DIRECT mode file. For BINARY files, READ (F, X) is equivalent to:

```
BEGIN
   X := F^;
   GET (F)
END
```

## 16.2.2 READ Formats

The READ process for formatted types (everything except CHAR, STRING, and LSTRING) first reads characters into an internal LSTRING and then decodes the string to get the value.

Three important points apply to formatted reads:

1. Leading spaces, tabs, form feeds, and line markers are skipped. For example, when doing READLN (I, J, K) where I, J, and K are integers, the numbers can all be on the same line or spread over several lines.
2. Characters are read as long as they are in the set of characters valid for the type wanted. For example, "-1-2-3" is read as the string of characters for a single INTEGER, but gives an error when the string is decoded. This means that items should be separated by spaces, tabs, line markers, or characters not permitted in the format.

3. M and N values in READ are ignored, except as noted for an N value with enumerated types. M and N parameters are not accepted in BINARY reads.

Most of the formatting rules below apply to the function DECODE, as well.

1. INTEGER and WORD types

   If P is of type INTEGER, WORD, or a subrange thereof, then READ (F, P) implies reading a sequence of characters from F which form a number according to the normal Pascal syntax, and then assigning the number to P. Nondecimal notation (16#C007, 8#74, 10#19, 2#101, #Face) is accepted for both INTEGER and WORD, with a radix of 2 through 36. If P is of an INTEGER type, a leading plus (+) or minus (-) sign is accepted. If P is of a WORD type, then numbers up to MAXWORD are accepted (32768 .. 65535).

2. REAL and INTEGER4 types

   If P is of type REAL, or at the extend level type INTEGER4, READ (F, P) implies reading a sequence of characters from F that form a number of the appropriate type and assigning the number to P. Nondecimal notation is not accepted for REAL numbers, but is accepted for INTEGER4 numbers. When reading a REAL value, a number with a leading or trailing decimal point is accepted, even though this form gives a warning if used as a constant in a program.

3. Enumerated and Boolean types

   At the extend level, if P is an enumerated type or BOOLEAN, a number is read as a WORD subrange and a value assigned to P such that the number is the ORD of the enumerated type's value. In addition, if P is of type BOOLEAN, reading one of the character sequences 'TRUE' or 'FALSE' cause true and false, respectively, to be assigned to P. The number read must be in the range of the ORD values of the variable.

   Also at the extend level, if the parameter P is an enumerated type and includes the :N notation as in READ (P::N), characters are read from the file F that form a valid identifier or number. If the characters form a number it is assumed to be the ORD value described in the previous paragraph, and if the characters form an identifier that is one of the enumerated type's constant identifiers, its value is assigned to P. In addition, if the variable is BOOLEAN, reading one of the digits 1 or 0 causes either true or false to be assigned to the BOOLEAN variable. 'TRUE' and 'FALSE' are also accepted as the BOOLEAN constant identifiers.

The actual value of N is ignored: using the N notation directs the compiler to save the enumerated type's constant identifiers and make them available to the applicable READ routine. Omitting the N notation saves memory that would be used for the identifiers.

4. Reference types

   At the extend level, if P is a pointer type, a number is read as a WORD and assigned to P, in a way that depends on your implementation, so that writing a pointer and later reading it yields the same pointer value. The address types should be read as WORDs using (.R) or (.S) notation.

5. String types

   At the extend level, if P is a STRING (n), then the next "n" characters are read sequentially into P. Preceding line markers, spaces, tabs, or form feeds are not skipped. If the line marker is encountered before "n" characters have been read, the remaining characters in P are set to blanks and the file position remains at the line marker.

   If the STRING is filled with n characters before the line marker is encountered, the file position remains at the next character. In a few implementations there may be a limit of 255 characters on the length of a STRING read. P can be the super array type STRING (e.g., a reference parameter or pointer referent variable).

   At the extend level, if P is an LSTRING (n), then the next "n" characters are read sequentially into P, and the length of the LSTRING is set to "n". Preceding line markers, spaces, tabs, or form feeds are not skipped. If the line marker is encountered before "n" characters have been read, the length of the LSTRING is set to the number of characters read and the file position remains at the line marker.

If the LSTRING is filled with "n" characters before the line marker is encountered, the file position remains at the next character. P can be the super array type LSTRING (e.g., a reference parameter or pointer referent variable). READ (LSTRING) is handy when reading entire lines from a textfile, especially when the length of the line is needed. For example, the easiest way to copy a textfile is by using READLN and WRITELN with an LSTRING variable.

Currently, READ and READLN do not use M field width parameters: you cannot read the line '123456' as two INTEGER numbers with READ (I:3, J:3). However, you can read two LSTRING (3) items and then decode them to achieve the same effect.

## 16.2.3 WRITE and WRITELN

### PROCEDURE WRITE
### PROCEDURE WRITELN

File system intrinsic procedures for textfile I/O. WRITE and WRITELN write textfiles. Both are defined in terms of the more primitive operation, PUT; that is, if P is an expression of type CHAR and F is a file of type TEXT, then WRITE (F, P) is equivalent to:

```
BEGIN
   F^ := P;
   {Assign P to buffer variable F^.}
   PUT (F)
   {Assign F^ to next component of file.}
END
```

WRITE can take more than one parameter, as in WRITE (F, P1, P2, . . . Pn). This is equivalent to the following:

```
BEGIN
     WRITE (F, P1);
     WRITE (F, P2);
        .
        .
     WRITE (F, Pn)
END
```

The procedure WRITELN writes a line marker to the end of a line. In all other respects, WRITELN is analogous to WRITE. Thus, WRITELN (F, P1, P2, . . . Pn) is equivalent to:

```
BEGIN
    WRITE (F, P1, P2, . . . Pn);
    WRITELN (F)
END
```

If either WRITE or WRITELN has no file parameter, the default file parameter is OUTPUT. Therefore, the first statement in each of the following pairs is equivalent to the second:

```
WRITE (P1, P2, . . . Pn)
WRITE (OUTPUT, P1, P2, . . . Pn)

WRITELN (P1, P2, . . . Pn)
WRITELN (OUTPUT, P1, P2, . . . Pn )
```

At the standard level, parameters in a WRITE can be expressions of any of the following types:

CHAR

INTEGER

REAL

BOOLEAN

STRING

At the extend level, expressions can also be of the following types:

WORD

INTEGER4

LSTRING

an enumerated type

a pointer type

Parameters may take optional M and N values (see Section 16.2.4, "WRITE Formats," for information about M and N parameters).

Although the procedure WRITE can also write to a BINARY file (i.e., not a textfile), this is not recommended for DIRECT files after a SEEK operation, because the complementary READ form does not work as you might expect.

For BINARY files, WRITE (F, X) is equivalent to:

```
BEGIN
   F^:= X;
   PUT (F)
END
```

The form WRITE (F, P1, P2, . . . Pn) is also acceptable. Normally, BINARY writes do not accept M and N values.

## 16.2.4 WRITE Formats

In textfiles, data parameters to WRITE and WRITELN may take one of the following forms:

```
P   P:M   P:M:N   P::N
```

The M and N values can be considered value parameters of type INTEGER and are used for formatting in various ways. The extend level permits M and N values for both READs and WRITEs, and permits giving N without M, as in:

```
P::N
```

Using them in a nonstandard way is an error not caught at the standard level. In some cases only M, or N, or neither, is actually used; unused M and N values are ignored.

Omitting M or N is the same as using the value MAXINT. For example, WRITE (12 : MAXINT) uses the default M value (8 in this case). Currently, M and N values are not accepted for BINARY files. In WRITE, the M value is the field width used as the number of characters to write. In ISO-Pascal, M must be greater than zero, and if the expression being written requires less than M characters, then it is padded on the left with spaces.

At the extend level, M can also be negative or zero. If it is negative, the absolute value of M is used, but padding of spaces occurs on the right instead of the left. If it is zero, no characters are written. These are ISO standard errors not caught in MS-Pascal.

If the representation of the expression cannot fit in ABS (M) character positions, then extra positions are used as needed for numeric types, or the value is truncated on the right for string types. If M is omitted or equal to MAXINT, a default value is used.

The N value signifies:

1. the number of decimal places if P is of type REAL
2. the output radix if P is of type INTEGER, WORD, INTEGER4, or pointer
3. the numeric or identifier value if P is of an enumerated type

Most of the following formatting rules apply to the function ENCODE as well.

1. INTEGER and WORD types

   If P is of type INTEGER, WORD, or a subrange thereof, then the decimal representation of P is written on the file. If P is a negative INTEGER, a leading minus sign is always written. WORD values are never negative. For INTEGER and WORD values, the default M value is 8.

   If ABS (M) is smaller than the representation of the number, additional character positions are used as needed. N is used to write in hexadecimal, decimal, octal, binary, or other base numbering using N equal to a number from 2 to 36; this is an extension to the ISO standard. If N is not 10 (or omitted or MAXINT), then padding on the left is with zeros and not spaces. Omitting N or setting N to MAXINT or 10 implies a decimal radix.

   WORD decimal numbers from 32768 to 65535 are written normally and not in their negative integer equivalents. All values written should be separated by spaces or some other character not valid in numbers, so that values are read as separate numbers.

2. REAL and INTEGER4 types

   If P is of type REAL, a decimal representation of the number P, rounded to the specified number of decimal places, is written on the file. If the N is missing or equal to

MAXINT, a floating-point representation of P is written to the file, consisting of a coefficient and a scale factor. If N is included, a rounded fixed-point representation of P is written to the file, with N digits after the decimal point. If N is zero, P is written as a rounded integer, with a decimal point. The default value of M for REAL values is 14.

Some examples of WRITE operations on REAL values:

| **This Statement** | **Produces This Output** |
|---|---|
| `WRITE (123.456)` | `' 1.2345600E+02'` |
| `WRITE (123.456:20)` | `' 1.2345600000000E+02'` |
| `WRITE (123.456::3)` | `'        123.456'` |
| `WRITE (123.456:2:3)` | `' 123.456'` |
| `WRITE (123.456:-20:3)` | `'123.456` |

At the extend level, if P is of type INTEGER4, the decimal representation of P is written on the file. The N value is used to set the radix, as in type INTEGER. The default M value is 14.

3. Enumerated and Boolean types

   At the extend level, if P is an enumerated type and N is omitted or equal to MAXINT then ORD (P) is written on the file, as if it were a WORD. If N is given with the value 1, the enumerated type's constant identifier for the value of P is written on the file, as if it were a STRING. Note that using this N notation causes memory to be allocated for the enumerated type's constant identifiers.

   At the standard level, if P is of type BOOLEAN, then one of the strings 'TRUE' or 'FALSE' is written to the file as a STRING. The ORD value is never written for BOOLEAN types as it is for enumerated types (although you can use WRITE(ORD(P)) instead).

4. Reference types

   At the extend level, if P is a pointer type, then P is written as a WORD. This is done in an implementation defined way such that writing a pointer and later reading it produces the same pointer value. The address types should be written as WORD values using (.R) or (.S) notation.

5. String types

   If P is of type STRING (n), then the value of P is written on the file. The default value of M is the length of the STRING, "n". If ABS (M) is less than the length of the string, then only the first ABS (M) characters are written. If M is zero, nothing is written. The right portion of the STRING is always truncated, even if M is negative. In a few implementations, there may be a limit of 255 characters on the length of a STRING write.

   At the extend level, if P is of type LSTRING (n), then the value of P is written on the file. The default value of M is the current length of the string, P.LEN. If ABS (M) is less than the current length, then only the first ABS (M) characters are written. If M is zero, then nothing is written. The right portion of the LSTRING is always truncated, even if M is negative. If ABS (M) is greater than the current length, spaces, not characters, fill the remaining positions past the length in the LSTRING. Note that a string of M blanks can be written with NULL : M.

## 16.3 Extend Level I/O

At the extend level, MS-Pascal has these additional I/O features:

1. You can access three FCB fields: F.MODE, F.TRAP, and F.ERRS.
2. A number of additional procedures are predeclared.
3. Temporary files are available.

Section 8.6, "File I/O: Extend Level," discusses FCB fields in the context of files. The additional procedures and temporary files are described in the following sections.

## 16.3.1 Extend Level Procedures

**PROCEDURE ASSIGN (VAR F; CONSTS N : STRING);**

A file system procedure for extend level I/O. Assigns an operating system filename in a STRING (or LSTRING) to a file F. The filename format depends on the target operating system. As a rule, ASSIGN truncates any trailing blanks. ASSIGN overrides any filename set previously. A filename must be set before the first RESET or REWRITE on a file. ASSIGN on an open file (after RESET or REWRITE but before CLOSE) produces an error. ASSIGN to INPUT or OUTPUT is allowed, but since these two files are opened automatically, they must be closed before being assigned to.

**PROCEDURE CLOSE (VAR F);**

A file system procedure for extend level I/O. Performs an operating system close on a file, ensuring that the file access is terminated correctly. This is especially important for file variables allocated on the stack or the heap. Since these files must be closed before a RETURN or DISPOSE loses the file control block, they are closed automatically when a RETURN or DISPOSE releases stack or heap file variables.

File variables with the STATIC attribute in procedures and functions are also closed automatically when the procedure or function returns. Files allocated statically at the program, module, or implementation level are automatically closed when the entire program terminates.

If necessary, when a CLOSE is executed, a file being written to has its operating system buffers flushed. However, the MS-Pascal buffer variable is not PUT. If a file of type TEXT is being written and the last nonempty line does not end with a line marker, one is added to the end of the last line. If the file has the mode SEQUENTIAL and is being written, an end-of-file is written.

Note that some runtime errors may remove control from the MS-Pascal runtime system. In these cases, files being written may not be closed, and the information in them may be lost. A CLOSE on a file that is already closed or never opened (no RESET or REWRITE) is permitted. CLOSE is not ignored if error trapping is on and there was a previous error. CLOSE turns off error trapping for the file, and clears the error status if no errors were found.

## PROCEDURE DISCARD (VAR F);

A file system procedure for extend level I/O. Closes and deletes an open file. DISCARD is much like CLOSE except that the file is deleted.

## PROCEDURE READFN (VAR F ; P1, P2, . . . PN);

A file system procedure for extend level I/O. READFN is the same as READ (not READLN) with two exceptions:

1. File parameter F should be present (INPUT is assumed, but a warning is given if F is omitted).
2. If a parameter P is of type FILE, a sequence of characters forming a valid filename is read from F and assigned to P in the same manner as ASSIGN.

Parameters of other types are read in the same way as the READ procedure.

Note that READFN is like READ, not like READLN, and does not read the trailing line marker. If the first parameter in a READFN call is a file of any type, it is assumed to be the textfile from which characters are read. It is not assumed that the file's name should be read using INPUT as the default source.

READFN is used internally to read a program's parameters. It is useful when reading a filename and assigning the filename to some file in one operation.

## PROCEDURE READSET (VAR F; VAR L : LSTRING, CONST S : SETOFCHAR);

A file system procedure for extend level I/O. READSET reads characters and puts them into L, as long as the characters are in the set S and there is room in L. If no file parameter is given, INPUT is assumed, as in READ and WRITE. Leading spaces, tabs, form feeds, and line markers are always skipped.

Reading ceases at the first line marker, which is never in the type CHAR.

READSET, along with ENCODE, is used by the runtime system to do the formatted READ procedures, as well as to read filenames with READFN. It is handy when reading and parsing input lines for simple command scanners.

In a segmented memory environment, the L and S parameters must reside in the default data segment.

## PROCEDURE SEEK (VAR F; N : INTEGER4);

A file system procedure for extend level I/O. In contrast to normal sequential files, DIRECT files are random access structures. SEEK is used to randomly access components of such files. To use a DIRECT file, the MODE field must be set to DIRECT before the file is opened with RESET or REWRITE; the file, F, must be a DIRECT mode file.

If the file is actually read or written sequentially, the usual READ and WRITE procedures can be used.

SEEK modifies a field in file F so that the next GET or PUT applies to record number N. The record number parameter N can be of type INTEGER or WORD, as well as of type INTEGER4. For textfiles (ASCII structure), records are lines; for other files (BINARY structure), records are components. Record numbers start at one (not zero). If F is an ASCII file, SEEK sets the lazy evaluation status "empty." If F is a BINARY file, SEEK waits for I/O to finish and sets the concurrent I/O status "ready."

SEEK is best illustrated by some examples. Assume for instance, that a BINARY structured, DIRECT mode file contains the following CHAR contents:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | |
| N = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

An implicit SEEK 1 is done after a REWRITE or a RESET. Thus, with DIRECT mode files, the following sequences of commands might be given:

```
RESET (F);
{Initial SEEK 1, followed by GET;}
{F^ now holds 'A'.}
SEEK (F, 5);
{File position set to 5; F^ still holds 'A'.}
C := F^
{C is now equal to 'A'; C does not equal 'E'.}
```

Note that the fifth component is not assigned to C, as you might expect. To obtain this value, the following sequences of commands should be executed:

```
RESET (F);
{Initial SEEK 1, followed by GET;}
{F^ now holds 'A'.}
SEEK (F, 5);
{File positioned at 5.}
GET (F);
{File buffer variable is loaded with 'E'.}
C := F^
{C gets value 'E'.}
```

The rule to follow is to always follow a SEEK (F, N) with a GET to assure that the nth component is contained in the buffer variable.

GET and PUT operate normally on DIRECT mode files with BINARY structured files. However, READ and WRITE work only with ASCII files, i.e., textfiles. READ, in particular, will not work with DIRECT mode BINARY files, because it assigns the buffer variable's value before it performs a GET. On the other hand, GET and PUT are not normally used with ASCII structured DIRECT mode files. Lazy evaluation makes READ and WRITE more appropriate. Care should always be taken when mixing normal sequential operations with DIRECT mode SEEK operations.

## 16.3.2 Temporary Files

Sometimes a program needs a "scratch" file for temporary, intermediate data. If this is the case, you may create a temporary file that is independent of the operating system. To do so, without having to give the file a name in a specific format, ASSIGN a zero character as the name of the file. For example:

```
ASSIGN (F, CHR (0))
```

The file system creates a unique name for the file when it sees that the zero character has been assigned as a name.

In environments where several running jobs are sharing a file directory, the job number is usually part of the name. Temporary files are deleted when they are closed, either explicitly or when the file gets deallocated. RESET and REWRITE do not delete the file.

# Chapter 17
# Compilable Parts of a Program

The Microsoft Pascal Compiler can compile three kinds of source files: programs, modules, and implementations of units. Modules and implementations of units can be compiled separately and later be linked to a program without recompilation. At the standard level, you may compile only entire programs; modules and units are MS-Pascal features available at the extend level.

Example of a compilable program:

```
PROGRAM MAIN (INPUT, OUTPUT);
BEGIN
  WRITELN ('Main Program')
END. {Main}
```

Example of a compilable module:

```
MODULE MOD_DEMO;
{No parameter list in heading}
    PROCEDURE MOD_PROC;
    BEGIN
        WRITELN
            ('Output from MOD_PROC in MOD_DEMO.')
    END
END. {Mod_Demo}
```

Example of a compilable unit:

```
INTERFACE;
  UNIT UNIT_DEMO (UNIT_PROC);
  {UNIT_PROC is the only exported identifier.}
  PROCEDURE UNIT_PROC
END
IMPLEMENTATION OF UNIT_DEMO;
  PROCEDURE UNIT_PROC
  BEGIN
     WRITELN
         ('Output from UNIT_PROC in UNIT_DEMO.')
  END
END. {Unit_Demo}
```

If you compile MODULE MOD_DEMO AND UNIT UNIT_DEMO separately, you can later incorporate them into the main program as shown below:

```
{INTERFACE required at the start of any }
{source that implements or uses a unit.}

INTERFACE;
  UNIT UNIT_DEMO (UNIT_PROC);
  PROCEDURE UNIT_PROC
END;

PROGRAM MAIN (INPUT, OUTPUT);
{USES clause below needed to connect}
{implementation and program.}
USES UNIT_DEMO;

{EXTERN declaration needed to connect}
{module's procedure.}
PROCEDURE MOD_PROC; EXTERN;
BEGIN
   WRITELN ('Output from Main Program.');
   MOD_PROC;
   UNIT_PROC
END. {End of main program.}
```

When the program MAIN is compiled, the output consists of the following pieces:

1. output from program MAIN
2. output from MOD_PROC declared in MOD_DEMO
3. output from UNIT_PROC declared in UNIT_DEMO

The rules governing the construction and use of programs, modules, and units are discussed in the following sections:

Section 17.1, "Programs"

Section 17.2, "Modules"

Section 17.3, "Units"

## 17.1 Programs

Except for its heading and the addition of a period at the end, a Pascal program has the same format as a procedure declaration. The statements between the keywords BEGIN and END are called the body of the program.

Example of a program:

```
{Program heading}
PROGRAM ALPHA (INPUT, OUTPUT, A_FILE, PARAMETER);

{Declaration section}
VAR A_FILE : TEXT; PARAMETER : STRING (10);

{Program body}
BEGIN
   REWRITE (A_FILE);
   WRITELN (A_ FILE, PARAMETER)
END.
{Ends with period (.)}
```

The word "ALPHA" following the reserved word "PROGRAM" is the program identifier. The program identifier becomes the identifier for a parameterless PUBLIC procedure, at a scope above all other identifiers in the program. This procedure also has the PUBLIC identifier ENTGQQ, which is called during initialization to start program execution.

You could call the program body as a PUBLIC procedure from another program, or from a module or unit, using the program identifier or ENTGQQ as the procedure name (but doing so is not recommended). This means that you can redeclare the program identifier within a program, and the usual scoping rules apply. The program identifier is at the same level as the predeclared identifiers, so giving a program an identifier like INTEGER or READ generates an error message.

The program parameters denote variables that are set from outside the program. The program communicates with its environment through these variables.

At the standard level, all variables of any FILE type should be present as program parameters, since there is no other way to give an operating system filename to the file. However, at the extend level, you may use the ASSIGN and READFN procedures to assign filenames, so file variables need not appear as program parameters.

Program parameters differ entirely from procedure parameters; they are not passed as parameters to the procedure that is the body of the program. All program parameters must be declared in the variable declaration part of the block constituting the program. If there are no program parameters and the files INPUT and OUTPUT are not referenced, use the following form instead:

```
PROGRAM identifier;
```

The two standard files INPUT and OUTPUT receive special treatment as program parameters. Their values are not set like other program parameters and should not be declared, since they are already predeclared. Each should be present as a program parameter if used either explicitly or implicitly in the program:

```
WRITE (OUTPUT, 'Prompt: ');  {Explicit use}
READLN (INPUT, P);

WRITE ('Prompt: ')           {Implicit use}
READLN (P);
```

The compiler gives a warning if you use them in the program but omit them as program parameters. The only effect of INPUT and OUTPUT as program parameters is to suppress this warning.

You may redefine the identifiers INPUT and OUTPUT. However, all textfile input and output procedures and functions (READ, EOLN, etc.) still use the original definition. RESET (INPUT) and REWRITE (OUTPUT) are generated automatically, whether or not they are present as program parameters; you may also generate them explicitly.

Program initialization gives a value to every program parameter variable, except INPUT and OUTPUT. Each parameter must be either of a simple type or of a STRING, LSTRING, or FILE type (i.e., any type accepted by the READFN procedure). Program parameters must be entire variables: no component selection is permitted.

Internally, each program parameter uses the file INPUT and generates READFN calls. Before each parameter is read, a special call is made to the internal routine PPMFQQ. PPMFQQ gets characters returned from an operating system interface routine called PPMUQQ, which gets them from the command line. PPMFQQ then effectively puts those characters at the start of the file INPUT. The identifier of the parameter is passed to both routines (PPMFQQ and PPMUQQ). Some operating systems then use the identifier as a prompt.

The use of program parameters in MS-Pascal can best be illustrated by showing how to change a program into a procedure. Suppose you have a program like the following:

```
PROGRAM ALPHA (INPUT, OUTPUT, P1, P2, . . . Pn);
declarations
{Including those for P1, P2, . . . Pn}
BEGIN
   body
END.
```

PROGRAM ALPHA could then become the following procedure:

```
PROCEDURE ENTGQQ [PUBLIC];
declarations
{Including those for P1 , P2, . . . Pn}
BEGIN
   PPMFQQ ('P1'); READFN (INPUT, P1);
   PPMFQQ ('P2'); READFN (INPUT, P2);
   .
   .
   PPMFQQ ('Pn'); READFN (INPUT, Pn);
   PPMEQQ
   {Called after all parameters are read}
   program statements
END;
```

The action of the interface routine PPMFQQ depends on the target operating system.

Some operating systems have elaborate mechanisms to handle this kind of parameter, using menus and default values. If your system falls into this category, the same mechanism generally applies to MS-Pascal program parameters.

Other less sophisticated operating systems pass to a program the remainder of the command line that invoked it; in this case, parameter values are read from the command line.

If the operating system does not provide a program parameter mechanism, or if an error occurs while using such a mechanism, or if it does not supply enough parameter values, then the PPMFQQ routine reverts to handling parameter values itself. It prompts you for every parameter with the parameter's identifier and reads the value you give it for the parameter.

## 17.2 Modules

Modules provide a simple, straightforward method for combining several compilable segments into one program. Units, described in Section 17.3, "Units," provide a more powerful and structured method for achieving the same end.

Basically, a module is a program without a body. The identifier in the module heading has the same scope as a program identifier. The heading can also include attributes that apply to all procedures and functions in the module. There are no module parameters; nor is there a module body. A module ends with the reserved word END and a period.

Example of a module:

```
MODULE BETA [PUBLIC]; {Optional attributes}

PROCEDURE GAMMA;
    BEGIN WRITELN ('Gamma') END;

FUNCTION DELTA : WORD;
    BEGIN DELTA := 123 END;

END. {No body before END}
```

After the module identifier, you may give one or more attributes (in brackets) to apply to all of the procedures and functions nested directly in the module. Depending on which, if any, attributes you specify, the following assumptions or restrictions apply:

1. If there is no attribute list at all, the PUBLIC attribute is assumed. However, if a list is present but empty, PUBLIC is not assumed.
2. The EXTERN directive used with a particular procedure or function overrides the PUBLIC attribute given (or assumed) for the entire module.
3. EXTERN and ORIGIN cannot be given as attributes for an entire module, although you may specify them for individual procedures and functions.
4. If PURE or INTERRUPT are used, the module must contain only functions for PURE and procedures for INTERRUPT.
5. PUBLIC is the default attribute for all procedures and functions. However, in some cases, a PUBLIC procedure call has more overhead than a purely local one. In other cases, the identifier of a local procedure may conflict with a global identifier passed to the linker. To avoid these problems, use PUBLIC with selected individual procedures and functions and empty brackets for the entire module (e.g., MODULE BETA [ ];).

Although a module contains no body, only declarations, you may use it as a parameterless procedure; that is, you may declare the module identifier as a procedure and call it from other programs, modules, or units. This module procedure (unlike a similar procedure for programs or units) is never called automatically, since there is no way for the compiler to know whether a module has been loaded and thus whether to generate a call to it.

However, in some cases, the compiler generates module initialization code that should be executed by calling the module as an EXTERN procedure. If such code is necessary, the compiler gives the warning:

```
Initialize Module
```

If you see this message, declare the module as a parameterless EXTERN procedure and call the procedure once before anything in the module is accessed. (You will need to do this if the module declares any FILE variables.)

Given a module M that declares its own file variables, a program that uses M should look like this:

```
PROGRAM P (INPUT, OUTPUT)
    .
    .
PROCEDURE M; EXTERN;
BEGIN
    M;  {Runtime call initializes}
    .   {file variables.}
    .
END.
```

If the module USES any interfaces that require initialization, the compiler generates a warning that you should declare the module EXTERN and call it as described in the previous paragraph.

If module M does not contain any of its own file variables or use any initialized units, there is no need to invoke M as a procedure in the body of the program or to declare it as an EXTERN procedure.

Variables within modules are not automatically given any attributes. Except for the initialization of FILE variables mentioned above, variables within modules are the same as program variables.

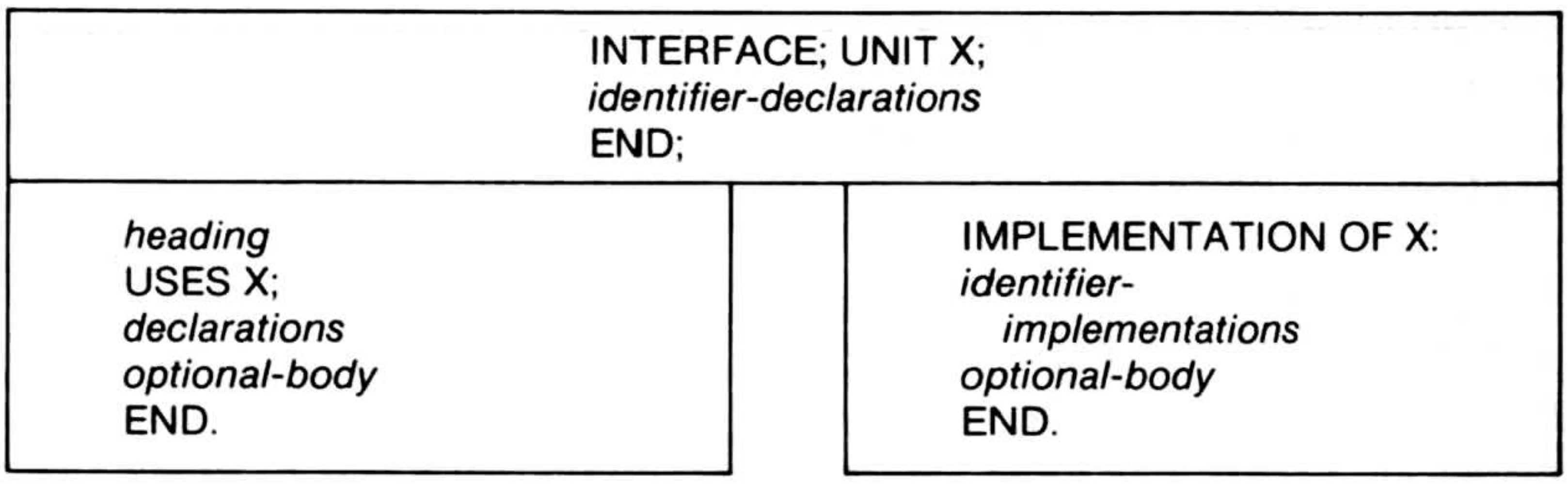
## 17.3 Units

MS-Pascal units provide a structured way to access separately compiled modules. A unit has two parts:

1. an interface
2. an implementation

The interface appears at the front of an implementation of a unit and at the front of any program, module, interface, or implementation that uses a unit.

A unit contains constants, types, super types, variables, procedures, and functions, all of which are declared in the interface of the unit. Any program, module, implementation, or another interface may use an interface. An implementation contains the bodies of the procedures and functions in a unit, as well as optional initialization for the unit. The general scheme is shown in Figure 17.1.



```
INTERFACE; UNIT X;
identifier-declarations
END;
```

```
heading
USES X;
declarations
optional-body
END.
```

```
IMPLEMENTATION OF X:
identifier-
   implementations
optional-body
END.
```

**Figure 17.1. A Microsoft Pascal Unit**

When you are using units, their interfaces go before everything else in a source file, either in an IMPLEMENTATION or in the program, module, or other unit that uses it. In the preceding diagram, the INTERFACE is shared; the same INTERFACE exists in both the IMPLEMENTATION source file and in the other source file. Conversely, any other program, module, or unit could USE UNIT X; similarly, there could be another IMPLEMENTATION OF X, in assembly language, for example.

By separating the interface from the implementation, you can write and compile a program before or while writing the implementation. Or, you may load a program with one of several implementations (for example, one in MS-Pascal or one in assembly language). A large MS-Pascal program is often better organized as a main program and a number of units (parts of the MS-Pascal runtime system are organized in this way). However, only a program, module, interface, or implementation can USE a unit, not an individual procedure or function.

A program, module, implementation, or interface that uses an interface must start with the source file for that interface. Generally, the interface source file is a separate file, and an $include metacommand at the start of the source file brings in the interface source itself at compile time. Because there is then only one master copy of the interface, this is easier and more reliable than physically inserting the interface everywhere it is used (and running the risk of ending up with several different versions).

In some applications, you may want several versions of the same interface. For example, there is a separate version of the MS-Pascal file control block interface for every target file system; the $included file is copied from the desired interface version before the program using it is compiled. Naturally, every version must declare the common identifiers; each might also have some constant values for use in $if metacommands for the version-specific portions of the interface.

Suppose the INTERFACE for UNIT X in Figure 17.1 is contained in the file X.INT. If that is so, the compiland using the unit and the IMPLEMENTATION of the unit need only to $include the interface file at the start of the source file, as shown in Figure 17.2.

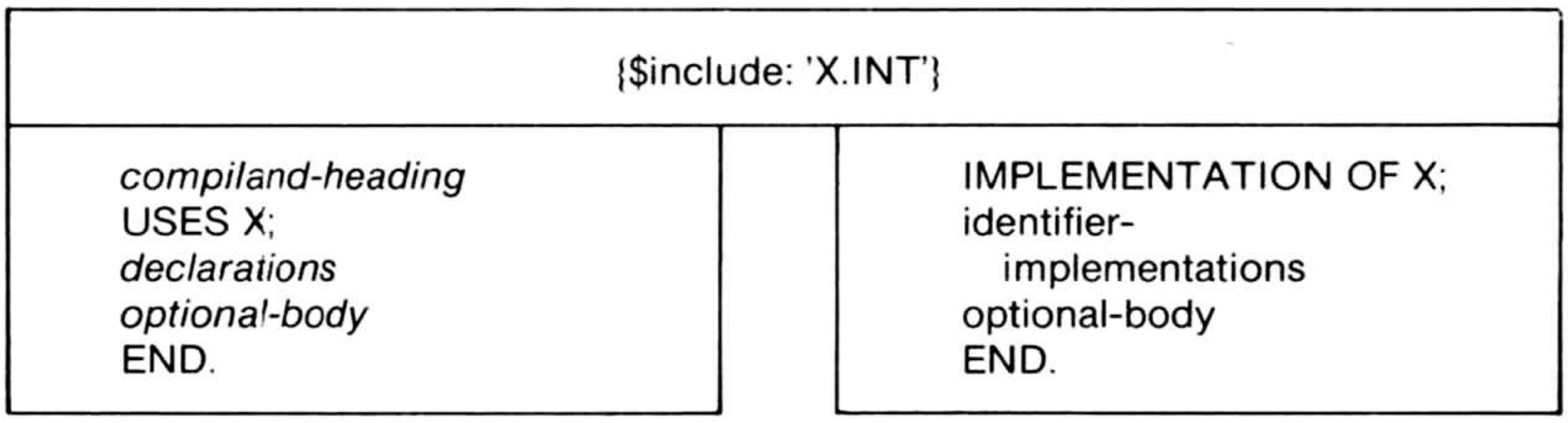


**Figure 17.2. Unit With File X.INT**

An MS-Pascal source file of any kind contains zero or more unit interfaces, separated by semicolons, and followed by a program, a module, or an implementation, which is followed by a period. Each of these entities is called a "division." See Section 17.3.1, "The Interface Division," and Section 17.3.2, "The Implementation Division," for details about divisions.

A unit consists of the unit identifier, followed by a list of identifiers in parentheses. These identifiers are called the constituents of the unit and are the ones provided by a unit or required by a program, module, or other unit. The unit is preceded by the keyword UNIT for a provided unit or USES for a required one.

All unit identifiers in a source file must be unique. The identifiers in parentheses, however, may differ in the providing and requiring divisions. Correspondence between identifiers provided and required is by position in the list (similar to formal and actual parameters in procedures).

The identifier list in a USES clause is optional; if not given, the identifiers in the UNIT list are used by default. Giving different identifiers in a USES clause allows you to change the identifiers in case several different interfaces have identifier conflicts. Multiple USES clauses can be combined; thus, the following statements are equivalent:

```
USES A; USES B; USES C;
USES A, B, C;
```

Note also that a unit may introduce optional initialization code. Such code is implied by the words BEGIN and END at the end of an interface and is provided in an optional body in an IMPLEMENTATION.

Example of a unit that introduces initialization code:

The program file, PLOTBOX:

```
{$include:'GRAPHI'}
PROGRAM PLOTBOX (INPUT, OUTPUT);
   USES GRAPHICS (MOVE, PLOT);
   {MOVE and PLOT are USEd identifiers.}
   BEGIN
       MOVE (0, 0);
       PLOT (10, 0); PLOT (10, 10);
       PLOT (0, 10); PLOT (0, 0)
   END.
```

The interface file, GRAPHI:

```
INTERFACE;
   UNIT GRAPHICS (BJUMP, WJUMP);
   {Exported identifiers are BJUMP and WJUMP.}
   {In the above PROGRAM, MOVE and PLOT}
   {are aliases for these identifiers.}
   PROCEDURE BJUMP (X, Y : INTEGER);
   PROCEDURE WJUMP (X, Y : INTEGER);
   {Procedure headings only above.}
BEGIN
{BEGIN implies initialization code.}
END;
```

The implementation file:

```
{$include:'GRAPHI'}
{$include:'BASEPL'}
{The following implementation USES}
{the UNIT BASEPL. Thus, the interface}
{is included above and the unit}
{used below.}
IMPLEMENTATION OF GRAPHICS;
{Implementation is invisible to user.}
   USES BASEPLOT;
    {Procedures BJUMP and WJUMP are}
    {implemented below.}
    {Note that only the identifiers}
    {are given in the heading.}
    {The parameter lists are given}
    {in the interface.}
   PROCEDURE BJUMP;
      BEGIN DRAWLINE (BLACK, X, Y) END;
   PROCEDURE WJUMP;
      BEGIN DRAWLINE (WHITE, X, Y) END;
BEGIN
{Begin initialization.}
   DRAWLINE (BLACK, 0, 0)
END.
```

The interface file, BASEPL:

```
INTERFACE;
   UNIT BASEPLOT (BLACK, WHITE, DRAWLINE);
     {Other identifiers besides procedure}
     {identifiers can be exported.}
     {Note that BLACK and WHITE are}
     {exported constant identifiers.}
   TYPE RAINBOW = (BLACK, WHITE, RED, BLUE, GREEN);
   PROCEDURE DRAWLINE (C : RAINBOW; H, V : INTEGER);
{No BEGIN; therefore, not an initialized unit.}
END;
```

A USES clause may occur only directly after a program, module, interface, or implementation heading. When the compiler encounters a USES clause, it enters each constituent identifier (from the UNIT clause or USES clause itself) in the symbol table. Identifiers for variables, procedures, and functions are associated with the corresponding identifiers in the interface, which then become external references for the linker.

If the sample program above were compiled, every reference to the procedure PLOT would generate an external reference to WJUMP. However, references to DRAWLINE would use the same identifier for the external reference.

Constants and types (including any super array types) in the interface are simply entered in the program's symbol table (along with the new identifier, if any). Thus, a type in an interface is identical to the corresponding type in the USES clause.

Record field identifiers are the same in the program, interface, and implementation. Enumerated type constant identifiers must be given explicitly, if needed; they are not automatically implied by the enumerated type identifier. Labels cannot be provided by an interface, since the target label of a GOTO must occur in the same division as the GOTO.

## 17.3.1 The Interface Division

The structure of an interface is as follows:

1. An interface section starts with the reserved word INTERFACE, an optional version number in parentheses, and a semicolon.
2. Next comes the keyword UNIT, the unit identifier, the parenthesized list of exported (constituent) identifiers, and another semicolon.
3. Any other units required by this interface come next, in USES clauses.
4. The last section is the actual declarations for all identifiers given in the interface list, using the usual CONST, TYPE, and VAR sections and procedure and function headings, in any order. No LABEL or VALUE sections are permitted.
5. The interface ends with BEGIN END if it has initialization, or just with END if it has no initialization.

Except for ORIGIN, which cannot currently be used in interfaces, most available attributes can be given to variables, procedures, and functions. Because the PUBLIC or EXTERN attribute or EXTERN directive is given automatically, you must not specify attributes that may conflict (e.g., PUBLIC and EXTERN).

Usually the only identifiers you declare are the constituents, but other identifiers are permitted. If the interface needs a call to initialize the unit, the keyword BEGIN generates the call. The interface ends with the reserved word END and a semicolon.

Example of an interface division:

```
INTERFACE (3);
   UNIT KEYFILE (FINDKEY, INSKEY, DELKEY, KEYREC);
      USES KEYPRIM (KFCB, KEYREC);

      PROCEDURE FINDKEY (CONST NAME : LSTRING;
             VAR KEY : KEYREC;
             VAR REC : LSTRING);
      PROCEDURE INSKEY (CONST REC : LSTRING;
             VAR KEY : KEYREC);
      PROCEDURE DELKEY  (CONST KEY : KEYREC);
      PROCEDURE NEWKEY (CONST KEY : KEYREC);
   BEGIN
   {Signifies initialized unit.}
   END;
```

In this example, KEYREC is part of the unit KEYPRIM, but is exported as part of the unit KEYFILE. KFCB is also part of the KEYPRIM unit, but is not exported by the KEYFILE unit. NEWKEY is defined in the interface, but not exported by the KEYFILE unit. This is permitted, but pointless, since NEWKEY is unknown even in the implementation of the unit.

Memory available at compile time limits the number of identifiers the compiler can process. This limit can be a problem if you have many interfaces, especially interfaces that use other interfaces. The symptom is the following error message:

```
Compiler Out Of Memory
```

The message occurs before the final USES clause in the program, module, or implementation you are compiling. The cure is to reduce the number of identifiers in interfaces USEd by other interfaces. For example, make a single interface that contains only types (and type-related constants) shared by your other interfaces, and only USE this interface in the others.

If you include any file variables in the interface, the unit must be initialized. The compiler does not give the usual warning,

```
Initialize Variable
```

when you declare a file in an interface. If your interface contains files, be sure to end it with BEGIN END so that it will be initialized.

## 17.3.2 The Implementation Division

You may compile an implementation of a unit separately from other programs, modules, or units, but you must compile it along with its interface.

The structure of an implementation is as follows:

1. An implementation of an interface starts with the reserved words IMPLEMENTATION OF, followed by the unit identifier and a semicolon.
2. Next comes a USES clause for units it needs only for its own use.
3. Then comes the usual LABEL, CONSTANT, TYPE, VAR, and VALUE sections and all procedures and functions mentioned as constituents (which must be in the outer block) or used internally, in any order.

VALUE and LABEL sections may appear in the implementation, but not in the interface.

Example of an implementation:

```
IMPLEMENTATION OF KEYFILE;
   USES KEYPRIM (KEYBLOCK, KEYREC);

   VAR KEYTEMP : KEYREC;

   PROCEDURE FINDKEY;
      BEGIN
          .
          {Code for FINDKEY}
          .
      END;

   PROCEDURE INKEY;
      BEGIN
          .
          {Code for INKEY}
          .
      END;
```

```
    PROCEDURE DELKEY;
      BEGIN
            .
          {Code for DELKEY}
            .
      END;
  BEGIN
    .
    {Any initialization code goes here.}
    .
  END.
```

Constants, variables, and types declared in the interface are not redeclared in the implementation. However, you may declare other "private" ones. Procedures and functions that are constituents of the unit do not include their parameter list (it is implied by the interface) or any attributes. (The PUBLIC attribute is implied, unless the EXTERN directive is given explicitly.)

All procedures and functions in the interface must be defined in the implementation. However, they can be given the EXTERN directive so that several implementations (or an implementation and assembly code) can implement a single interface. All procedures and functions with the EXTERN directive must appear first; the compiler checks for this and issues an error message if the EXTERN directive is missing or misplaced.

You may implement a unit in assembly language, in which case all variables, procedures, and functions should generate public definitions for the loader. You may also implement units in other programming languages, such as MS-FORTRAN, or in a mixture of languages. If the interface is not implemented in MS-Pascal, it must give the proper calling sequence attribute (and of course you must be familiar with calling sequences and internal representation of parameters).

Several MS-Pascal runtime units are implemented partially in MS-Pascal and partially in assembly language. As mentioned, any implementation section that does not implement all interface procedures and functions must, at the start of the implementation, declare such procedures and functions to be EXTERN.

An implementation, like a program, may have a body. The body is executed when the program that uses the unit is invoked, so any initialization needed by the unit can be done. This includes internal initialization, such as file variable initialization, as well as user initialization code. If the source file contains several units, each implementation body is called in the order its USES clause appears in the source file. However, initialization code for a unit is executed only once, no matter how many clauses refer to it.

The body, as in a program, is a list of statements enclosed with the reserved words BEGIN and END. At initialization time, the version number of the interface with which the implementation was compiled is compared against the version number of the interface with which the program was compiled. These must be the same. This checking prevents you from trying to run a program with obsolete implementations. If no version number is given, zero is assumed.

The keyword BEGIN before the final END indicates a unit with initialization. If the word BEGIN is omitted, the implementation must not have a body and no initialization takes place. Uninitialized units lack the following:

1. user initialization code
2. a guarantee of only one initialization
3. a version number check

The format for an initialized implementation of a unit is similar to a program:

```
IMPLEMENTATION OF unit-identifier
declarations
BEGIN
     body {Initialization code}
END.
```

The format for an uninitialized implementation of a unit is similar to a module:

```
IMPLEMENTATION OF unit-identifier
declarations
{No initialization code}
END.
```

If the implementation for an uninitialized unit declares any files or USES any interfaces that require initialization, the compiler warns you to initialize the implementation. Initialization is done automatically if you add the keyword BEGIN to both the interface and the implementation. As with a module, you can declare an uninitialized unit to be a procedure with the EXTERN attribute and then initialize it by calling it from the program.

# Chapter 18 Microsoft Pascal Metacommands

Metacommands make up the compiler control language. Metacommands are compiler directives that allow you to control such things as the following:

1. Microsoft Pascal language level
2. debugging and error handling
3. optimization level
4. use of the source file during compilation
5. listing file format

You can specify one or more metacommands at the start of a comment; you should separate multiple metacommands with either spaces or commas. Spaces, tabs, and line markers between the elements of a metacommand are ignored. Thus, the following are equivalent:

```
{$page:12}
{$page : 12}
```

To disable metacommands within comments, place any character that is not a tab or space in front of the first dollar sign, as shown:

```
{x$page:12}
```

You may change compiler directives during the course of a program; for example, most of a program might use $list-, with a few sections using $list+ as needed. Some metacommands, such as $linesize, normally apply to an entire compilation.

If you are writing Microsoft Pascal programs for use with other compilers, keep in mind the fact that metacommands are always nonstandard and rarely transportable.

Metacommands invoke or set the value of a metavariable. Metavariables are classified as typeless, integer, on/off switch, or string.

1. Typeless metavariables are invoked when used, as in $extend.
2. Integer metavariables can be set to a numeric value, as in $page:101.
3. On/off switches can be set to a numeric value so that a value greater than zero turns the switch on and a value equal or less than zero turns it off, as in $mathck:1.
4. String metavariables can be set to a character string value, such as with $title:'com program'.

Table 18.1 illustrates the notational conventions observed in the metacommand descriptions that follow.

**Table 18.1**

**Metacommand Notation**

| Notation | Meaning |
|---|---|
| | Metacommand is typeless. |
| + or - | Metacommand is an on/off switch.<br>+ sets value to 1 (on).<br>- sets value to 0 (off).<br>Default is indicated by + or - in heading. |
| :*n* | Metacommand is an integer. |
| :*'text'* | Metacommand is a string. |

String values in the metalanguage may be either a literal string or string constant identifier. Constant expressions are not allowed for either numbers or strings, although you can achieve the same effect by declaring a constant identifier equal to the expression and using the identifier in the metacommand.

In metacommands only, Boolean and enumerated constants are changed to their ORD values. Thus, a Boolean false value becomes 0 and true becomes 1.

A complete alphabetical listing of MS-Pascal metacommands is given in Appendix G, "Summary of Microsoft Pascal Metacommands."

## 18.1 Language Level Setting and Optimization

The metacommands shown in Table 18.2 let you control the level (standard, extend, or system) at which the compiler processes your program and the degree to which optimization is used. Some of these metacommands may not be implemented in your version of the compiler.

**Table 18.2**

**Language and Optimization Level**

| Name | Description |
|---|---|
| $decmath | Directs the compiler to use the decimal math routines in the auxiliary decimal math runtime library. |
| $extend | Adds extend level features |
| $floatcalls | Directs the compiler to make calls to the real number math routines; set to $floatcalls+ by default |
| $real:*n* | Sets the length of the REAL type |
| $rom | Gives a warning on static initialization |
| $simple | Disables global optimizations |
| $size | Minimizes size of code generated |
| $speed | Minimizes execution time of code |
| $standard | Enables standard level only |
| $system | Adds extend and system level features |

The compiler issues a warning message if it encounters a feature whose level is not enabled. The default setting is $extend, which permits structured extensions that are relatively safe and portable. It also requires you to explicitly request $system extensions, which are by their nature low level, machine dependent, and relatively unstructured.

$Integer and $real set the length (i.e., precision) of the standard INTEGER and real data types. $Integer takes an integer parameter, which must be 2 or 4. However, you may set $real to either 4, or 8 (the default), to make type REAL identical to REAL4 or REAL8, respectively.

The effect of the $size and $speed metacommands varies with the version of the optimizer in your implementation of the compiler. The default is $size. If you select $simple, no optimization of any kind is done. $Size, $speed, and $simple are all mutually exclusive. If $rom is set, the compiler gives a warning that static data will not be initialized in either of the following situations:

1. at a VALUE section
2. every place where static data initialization occurs due to $initck (described in Section 18.2, "Debugging and Error Handling")

$Decmath+ directs the compiler to make calls to the decimal math support routines in which decimal floating-point numbers up to 14 digits and within a limited exponent range can be represented exactly. The results of the operations on the numbers in this format are also represented exactly if they are in the allowable range. If $decmath+ is used, it must precede any program text and may not be switched on and off. (You must link DECMATH.LIB when using $decmath+ in your source code.)

$Floatcalls+ generates calls to a real math runtime package to carry out floating-point operations. The $floatcalls- metacommand directs the compiler to generate in-line instruction "skeletons" for floating-point operations.

## 18.2 Debugging and Error Handling

The metacommands shown in Table 18.3 are for debugging and error handling. They also generate code to check for runtime errors.

**Table 18.3**

**Debugging and Error Handling**

| Metacommand | Description |
|---|---|
| $brave+ | Sends error messages and warnings to the terminal screen |
| $debug- | Turns on or off all the debug checking (CK in metacommands below); off by default |
| $entry- | Generates procedure entry/exit calls for debugger |
| $errors:*n* | Sets number of errors allowed per page (default is 25) |
| $goto- | Flags GOTO statements as "considered harmful" |
| $indexck- | Checks for array index values in range, including super array indices; off by default |
| $initck- | Checks for use of uninitialized values; off by default |
| $line- | Generates line number calls for the debugger |
| $mathck- | Checks for mathematical errors such as overflow and division by zero; off by default |
| $nilck- | Checks for bad pointer values; off by default |
| $rangeck- | Checks for subrange validity; off by default |
| $runtime- | Determines context of runtime errors |
| $stackck- | Checks for stack overflow at procedure or function entry; off by default |
| $tagck- | Checks tag fields in variant records; off by default |
| $warn+ | Gives warning messages in listing file |

If any check is on when the compiler processes a statement, tests relevant to the statement are done. A runtime error invokes a call to the runtime support routine, EMSEQQ (synonymous with ABORT). When EMSEQQ is called, the compiler passes the following information to it:

1. an error message
2. a standard error code
3. an optional error status value, such as an operating system return code

EMSEQQ also has available:

1. the program counter at the location of the error
2. the stack pointer at the location of the error
3. the frame pointer at the location of the error
4. the current line number (if $line is on)
5. the current procedure or function name and the source filename in which the procedure or function was compiled (if $entry is on)

Each of these metacommands is discussed in more detail on the following pages. Most of the metacommands in this group may also be given as command line switches to the compiler. See Section 18.6, "Command Line Switches," for details.

**$brave+**

Sends error messages and warnings to your terminal (in addition to writing them to the listing file). If the number of errors and warnings is more than will fit on the screen, the earlier ones scroll off and you will have to check the listing file to see them all.

**$debug-**

Turns on or off all of the debug switches (i.e., those that end with "CK"). You may find it useful to use $debug- at the beginning of a program to turn all checking off and then selectively turn on only the debug switches you want. Alternatively, you may use this metacommand to turn all debugging on at the start and then selectively turn off those you don't need as the program progresses. By default, some error checks are on and some off.

**$entry-**

Generates procedure and function entry and exit calls. This lets a debugger or error handler determine the procedure or function in which an error has occurred. Since this switch generates a substantial amount of extra code for each procedure and function, you should use it only when debugging. Note that $line+ requires $entry+; thus, $line+ turns on $entry, and $entry- turns off $line.

**$errors:*n***

Sets an upper limit for the number of errors allowed per page. Compilation terminates if that number is exceeded. The default is 25 errors and/or warnings per page.

**$goto-**

Flags GOTO statements with a warning that they are "considered harmful." This warning may be useful in either of the following circumstances:

1. to encourage structured programming in an educational environment
2. to flag all GOTO statements during the process of debugging

**$indexck-**

Checks that array index values, including super array indices, are in range. Since array indexing occurs so often, bounds checking is enabled separately from other subrange checking.

## $initck-

Checks for the occurrence of uninitialized values, such as the following:

1. uninitialized INTEGERs and 2-byte INTEGER subranges with the hexadecimal value 16#8000
2. uninitialized 1-byte INTEGER subranges with the hexadecimal value 16#80
3. uninitialized pointers with the value 1 (if $nilck is also on)
4. uninitialized REALs with a special value

The $initck metacommand generates code to perform the following actions:

1. set such values uninitialized when they are allocated
2. set the value of INTEGER range FOR-loop control variables uninitialized when the loop terminates normally
3. set the value of a function that returns one of these types uninitialized when the function is entered

$Initck never generates any initialization or checking for WORD or address types. Statically allocated variables are loaded with their initial values. Also, $initck does not check values in an array or record when the array or record itself is used.

Variables allocated on the stack or in the heap are assigned initial values with generated code. $Initck does not initialize any of the following classes of variables:

1. variables mentioned in a VALUE section
2. variant fields in a record
3. components of a super array allocated with the NEW procedure

## $line-

Generates a call to a debugger or error handler for each source line of executable code. This allows the debugger to determine the number of the line in which an error has occurred. Because this metacommand generates a substantial amount of extra code for each line in a program, you should turn it on only when debugging. Note that $line+ requires $entry+, so $line+ turns on $entry, and $entry– turns off $line.

**$mathck-**

Checks for mathematical errors, including INTEGER and WORD overflow and division by zero. $Mathck does not check for an INTEGER result of exactly -MAXINT-1 (i.e., #8000); $initck does catch this value if it is assigned and later used.

Turning $mathck off does not always disable overflow checking. There are, however, library routines that provide addition and multiplication functions that permit overflow (LADDOK, LMULOK, SADDOK, SMULOK, UADDOK, and UMULOK). See Section 15.2, "Directory of Functions and Procedures," for descriptions of each of these functions.

**$nilck-**

Checks for the following conditions:

1. dereferenced pointers whose values are NIL
2. uninitialized pointers if $initck is also on
3. pointers that are out of range
4. pointers that point to a free block in the heap

$Nilck occurs whenever a pointer is dereferenced or passed to the DISPOSE procedure. $Nilck does not check operations on address types.

**$rangeck-**

Checks subrange validity in the following circumstances:

1. assignment to subrange variables
2. CASE statements without an OTHERWISE clause
3. actual parameters for the CHR, SUCC, and PRED functions
4. indices in PACK and UNPACK procedures
5. set and LSTRING assignments and value parameters
6. super array upper bounds passed to the NEW procedure

**$runtime-**

If the $runtime switch is on when a procedure or function is compiled, the "location of an error" is the place where the procedure or function was called rather than the location in the procedure or function itself. This information is normally sent to your terminal, but you could link in a custom version of EMSEQQ, the error message routine, to do something different (such as invoke the runtime debugger or reset a controller).

**$stackck-**

Checks for stack overflow when entering a procedure or function and when pushing parameters larger than four bytes on the stack. In some implementations, stack overflow is always checked. In some implementations, stack overflow is never checked in procedures with the INTERRUPT attribute.

**$tagck-**

Checks tag values when accessing a variant field. Only those tag fields with identifiers (i.e., whose value is actually stored in the record) are checked.

**$warn+**

Sends warning messages to the listing file (this is the default). If this switch is turned off, only fatal errors are printed in the source listing.

## 18.3 Source File Control

A small group of metacommands provide some measure of control over the use of the source file during compilation. These commands are listed in Table 18.4 and described in more detail below.

**Table 18.4**

**Source File Control**

| Name | Description |
|---|---|
| $if *constant*<br>$then *text1*<br>$else *text2*<br>$end | Allows conditional compilation of *text1* source if *constant* is greater than zero |
| $include:'*filename*' | Switches compilation from current source file to source file named |
| $inconst:*text* | Allows interactive setting of constant values at compile time |
| $message:'*text*' | Allows the display of a message to the terminal screen to indicate which version of a program is compiling |
| $pop | Restores saved value of all metacommands |
| $push | Saves current value of all metacommands |

Because the compiler keeps one look-ahead symbol, it actually processes metacommands that follow a symbol before it processes the symbol itself. This characteristic of the compiler can be a factor in cases such as the following:

```
CONST Q = 1;
{$if q $then}
{Q is undefined in the $if.}

CONST Q = 1; DUMMY = 0;
{$if q $then}
{Now Q is defined.}
X := P^;
{$nilck+}
{NILCK applies to P^ here.}

X := P^;;;
{NILCK doesn't apply to P.}
{$nilck-}
```

Each of these metacommands is discussed in more detail on the following pages.

## $if *constant* $then *text* $end

Allows for conditional compilation of a source text. If the value of the constant is greater than zero, then source text following the $if is processed; otherwise it is not. An $if $then $else construction is also available, as in the following example:

```
{$if msdos $then}
SECTOR = S12;
{$else}
SECTOR = S128;
{$end}
```

To simulate an "if not" construction, use the following form of the metacommand:

$if *constant* $else *text* $end

The constant may be a literal number or constant identifier. The text between $then, $else, and $end is arbitrary; it can include line breaks, comments, other metacommands (including nested $ifs), etc. Any metacommands within skipped text are ignored, except, of course, corresponding $else or $end metacommands.

Examples using the metaconditional:

```
{$if FPCHIP $then}
  CODEGEN (FADDCALL, T1, LEFTP)
{$end}
{$if COMPSYS $else}
  IF USERSYS THEN DOITTOIT
{$end}
```

## $include

Allows the compiler to switch processing from the current source to the file named. When the end of the file that was included is reached, the compiler switches back to the original source and continues compilation. Resumption of compilation in the original source file begins with the line of source text that follows the line in which the $include occurred. Therefore, the $include metacommand should always be last on a line.

## $inconst

Allows you to enter the values of the constants (such as those used in $ifs) at compile time, rather than editing the source. This is useful when you use metaconditionals to compile a version of a source for a particular environment, customer, target processor, etc. Compilation may be either interactive or batch oriented. For example, the metacommand $inconst:year produces the following prompt for the constant YEAR:

```
Inconst : YEAR =
```

You need only give a response like:

```
Inconst : YEAR = 1983
```

The response is presumed to be of type WORD. The effect is to declare a constant identifier named YEAR with the value 1983. This interactive setting of the constant YEAR is equivalent to the constant declaration:

```
CONST YEAR = 1983;
```

You may also respond with a quoted string literal to create a constant of type STRING (n). For example, the source file metacommand $inconst:header prompts for a header. By enclosing a literal string constant in quotes, you declare a string constant:

```
Inconst : HEADER = 'Processor Version 2.75'
```

## $message

Allows you to send messages to your terminal during compilation. This is particularly useful if you use metaconditionals extensively, for example, and need to know which version of a program is being compiled.

Example of the $message metacommand:

```
{$message:'Message on terminal screen!'}
```

## $push and $pop

Allow you to create a meta-environment you can store with $push and invoke with $pop. $Push and $pop are useful in $include files for saving and restoring the metacommands in the main source file.

## 18.4 Listing File Control

The metacommands listed in Table 18.5 and described in this section allow you to format the listing file as you wish.

**Table 18.5**

**Listing File Control Metacommands**

| Metacommand | Description |
|---|---|
| $linesize:*n* | Sets width of listing. Default is 79 or 131, depending on implementation. |
| $list+ | Turns on or off source listing. Errors are always listed. |
| $ocode+ | Turns on disassembled object code listing. |
| $page+ | Skips to next page. Line number is not reset. |
| $page:*n* | Sets page number for next page (does not skip to next page). |
| $pageif:*n* | Skips to next page if less than n lines left on current page. |
| $pagesize:*n* | Sets length of listing in lines. Default is 55. |
| $skip:*n* | Skips n lines or to end of page. |
| $subtitle:'*text*' | Sets page subtitle. |
| $symtab+ | Sends symbol table to listing file. |
| $title:'*text*' | Sets page title. |

### $linesize:*n*

Sets the maximum length of lines in the listing file. This value normally defaults to either 131 or 79, depending on the implementation.

### $list+

Turns on the source listing. Except for $list–, metacommands themselves appear in the listing. The format of the listing file is described in Section 18.5, "Listing File Format."

**$ocode+**

Turns on the symbolic listing of the generated code to the object listing file. Although the format varies with the target code generator, it generally looks like an assembly listing, with code addresses and operation mnemonics. In many cases, the identifiers for procedure, function, and static variables are truncated in the object listing file.

**$page+**

Forces a new page in the source listing. The page number of the listing file is automatically incremented.

**$page:*n***

Sets the page number of the next page of the source listing. $Page:*n* does not force a new page in the listing file.

**$pageif:*n***

Conditionally performs $page+, if the current line number of the source file plus *n* is less than or equal to the current page size.

**$pagesize:*n***

Sets the maximum size of a page in the source listing. The default is 55 lines per page.

**$skip:*n***

Skips *n* lines or to the end of the page in the source listing.

**$subtitle:'*subtitle*'**

Sets the name of a subtitle that appears beneath the title at the top of each page of the source listing.

**$symtab+**

If on at the end of a procedure, function, or compiland, sends information about its variables to the listing file (for example, see lines 14 and 17 in the sample listing file in Section 18.5, "Listing File Format"). The left columns contain the following:

1. the offset to the variable from the frame pointer (for variables in procedures and functions)
2. the offset to the variable in the fixed memory area (for main program and STATIC variables)
3. the length of the variable

A leading plus or minus sign indicates a frame offset. Note that this offset is to the lowest address used by the variable.

The first line of the $symtab listing contains the offset to the return address, from the top of the frame (zero for the main program), and the length of the frame, from the framepointer to the end including front end temporary variables. Code generator temporary variables are not included.

For functions, the second line contains the offset, length, and type of the value returned by the functions. The remaining lines list the variables, including their type and attribute keywords, as shown in the following list.

| **Keyword** | **Meaning** |
|---|---|
| Public | Has the PUBLIC attribute |
| Extern | Has the EXTERN attribute |
| Origin | Has the ORIGIN attribute |
| Static | Has the STATIC attribute |
| Const | Has the READONLY attribute |
| Value | Occurs in a VALUE section |
| ValueP | Is a value parameter |
| VarP | Is a VAR or CONST parameter |
| VarsP | Is a VARS or CONSTS parameter |
| ProcP | Is a procedural parameter |
| Segmen | Uses segmented addressing |
| Regist | Parameter passed in register |

**$title:'*title*'**

Sets the name of a title that appears at the top of each page of the source listing.

## 18.5 Listing File Format

The following discussion of listing file format is keyed to this sample listing:

```
Use   Title                                                   PAGE  1
User  Subtitle                                               12/11/82
                                                             10:49:17
JG IC Line#    Source Line     MS-Pascal Version 3.0        10/82
   00     1    PROGRAM foo;  {$symtab+}
   10     2    VAR i : integer; K : ARRAY [-9..0] OF integer,
          2    ----------------------Warning 156,Assumed ;^
   20     3    FUNCTION bar (VAR j : integer) : integer;
   20     4       VAR k : ARRAY [0 . . 9] OF integer;
   20     5       BEGIN
 + 21     6       GOTO 1;           {jump forward}
          6    ------------^Warning 281 Label Assumed Declared
  = 21    7         i  := bar (j);     {assign to global}
          8         1 :                 {label}
  / 21    9         j  := bar (i);     {global to VAR parm}
 -  21   10         GOTO 1;            {jump backward}
 *  21   11         RETURN; GOTO 1;{other jumps}
  % 21   12         i  := bar (i);     {other global reference}
    21   13         j  := bar(j);        {no global references}
    10   14       END;
         14    --------^306 Function Assignment Not Found

Symtab   14    Offset Length Variable - BAR
               -    2      24 Return offset, Frame length
               -    2       2 (func'n return) : Integer
               +    4       2 J               : Integer VarP
               -   22      20 K               : Array

    10   15    BEGIN
    11   16     i   := bar (i);
    00   17    END.

Symtab   17    Offset Length Variable
                    0      24 Return offset, Frame length
                    2       2 I               : Integer
                    4      20 K               : Array

               Errors  Warns In Pass One
                    1      2
```

Every page has a heading that includes such information as your title and subtitle, set with the metacommands $title and $subtitle, respectively. If these metacommands appear on the first source line, they take effect on the first page. The page number appears in the right side of the first line of the heading. In some versions, the date and time appear in the right side of the second and third line, respectively. You can set the page number with $page:*n* or start a new page with $page+.

The fourth line of the listing contains the column labels. The contents of the first three columns are as follows:

1. The JG column

   The JG column contains flag characters generated for your information. Jump flags, which appear under the J, may contain one of the following characters:

   | | |
   |---|---|
   | + | forward jump (BREAK or GOTO a label not yet encountered) |
   | - | backward jump (CYCLE or GOTO a label already encountered) |
   | * | other jumps (RETURN or a mixture of jumps) |

   Codes for global variables (not local to the current procedure or function) appear in the column under G:

   | | |
   |---|---|
   | = | assignment to a nonlocal variable |
   | / | passing a nonlocal variable as a reference parameter |
   | % | a combination of the two |

2. The IC column

   The IC column contains information about the current nesting levels. The digit under the I refers to the identifier (scope) level, which changes with procedure and function declarations, as well as with record declarations and WITH statements. The digit in the C column refers to the control statement level; this number changes with BEGIN and END pairs, as well as with CASE and END and REPEAT and UNTIL pairs. The number in this column is useful for finding missing END keywords. If a line is not actively used by the compiler, all these columns are blank. Thus you can locate a portion of the source accidentally commented out or skipped due to an $if and $end pair.

3. The Line column

   The Line column shows the line number of the line in the source file. An $included file gets its own sequence of line numbers. If $line is on, this line number and the source file name identify runtime errors.

Two kinds of compiler messages appear in the listing: errors and warnings. A compilation with any errors cannot generate code. A compilation with warnings only can generate code, but the result may not execute correctly. Warnings start with the word "Warning" and a number (see, for example, line 2 in the sample listing). Errors start with an error number (see line 14 in the sample listing).

You can suppress warning messages with the metacommand $warn–, but this is not generally recommended. The metacommand $brave+ sends error and warning messages to your terminal (as well as to the listing file). However, if there are more than will fit on a single screen, the first ones will scroll off.

The location of the error is indicated in the listing file with an up arrow (^). The message itself may appear to the left or right of the arrow and is preceded with a dashed line.

Sometimes, the compiler does not detect an error until after the listing of the following line. In this case, the error message line number is not in sequence. Tabs are allowed in the source and are passed on to the listing unchanged. If the tab spacing is every eight columns, the error pointer (^) is generally correct. However, an error pointer near the end of a line may be displaced if the following line has tabs.

If the compiler encounters an error from which it cannot recover, it gives the message "Compiler Cannot Continue!". This message appears if any of the following occurs:

1. The keyword PROGRAM (or IMPLEMENTATION, INTERFACE, or MODULE) is not found, or the program, module, or unit identifier is missing.
2. The compiler encounters an unexpected end-of-file.

3. The compiler finds too many errors; the maximum number of errors per page is set with the $errors metacommand (the default is 25).
4. The identifier scope becomes too deeply nested.

When the compiler is unable to continue, for whatever reason, it simply writes the rest of the program to the listing file with very little error checking.

# Appendices

# Appendix A
# Pascal Syntax Diagrams

The diagrams on the following pages show the fundamental syntax of the Microsoft Pascal language. They are arranged in the order that you would be likely to use the elements while writing a program. The meaning of the differently shaped outlines is as follows:

1. Ovals

   Indicate reserved words or symbols of the MS-Pascal language. These must be typed as shown.

2. Boxes

   Indicate higher-level constructions that usually have syntax diagrams of their own.

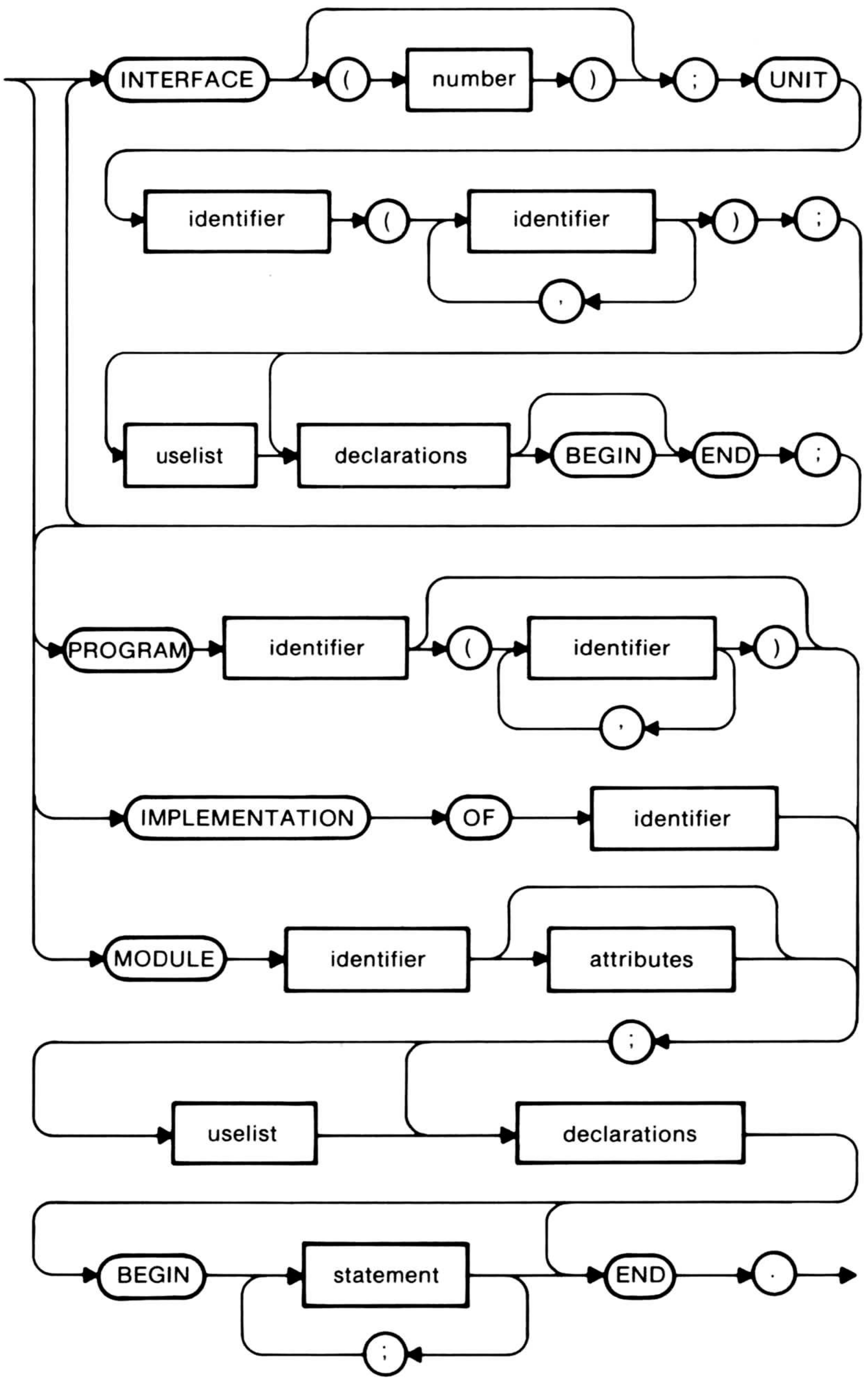
3. Circles

   Indicate punctuation that is required and must be typed as shown.

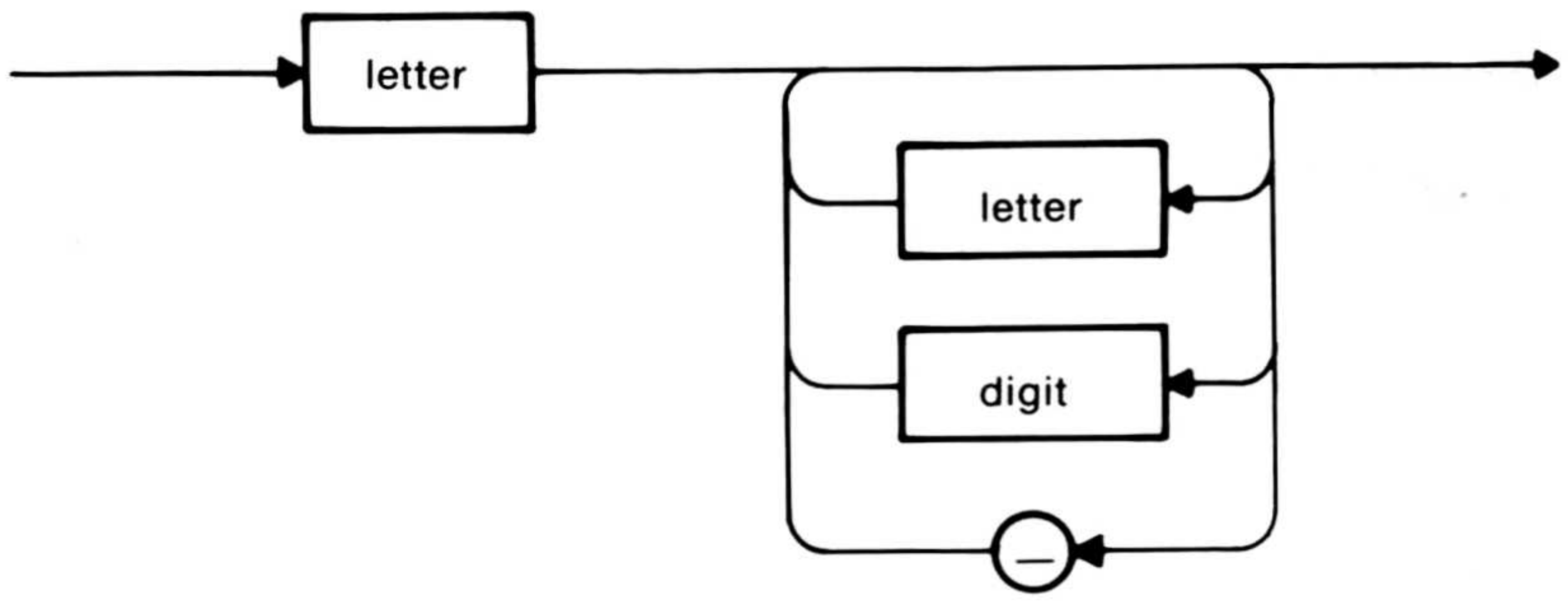
4. Arrows

   Help to show the path through the diagram, including any possible looping (i.e., repetition of syntax elements).

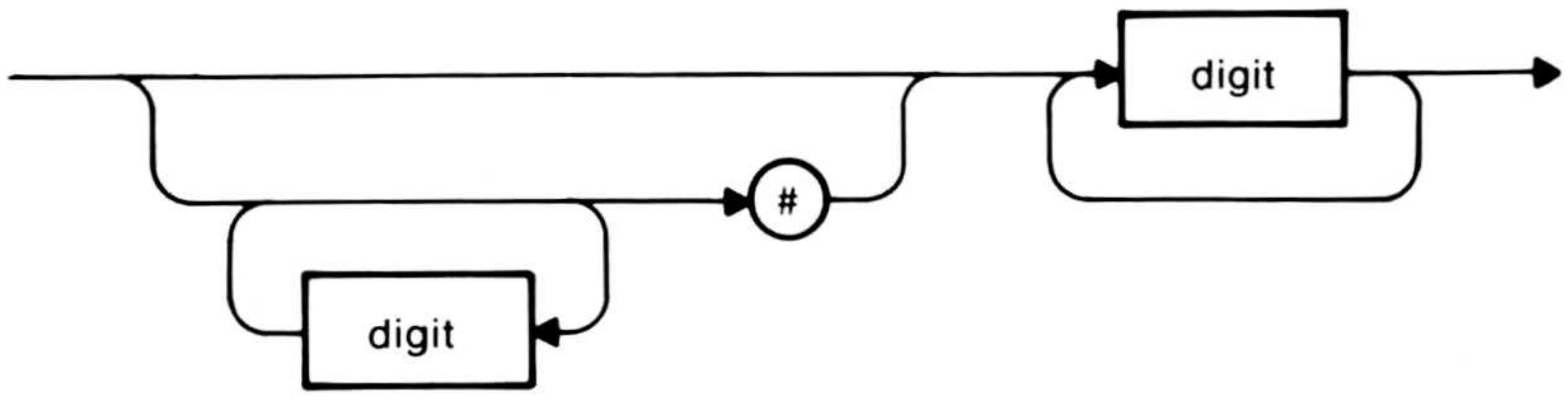
## Source File


INTERFACE
(
number
)
;
UNIT
identifier
(
identifier
,
)
;
uselist
declarations
BEGIN
END
;
PROGRAM
identifier
(
identifier
,
)
IMPLEMENTATION
OF
identifier
MODULE
identifier
attributes
;
uselist
declarations
BEGIN
statement
;
END
.

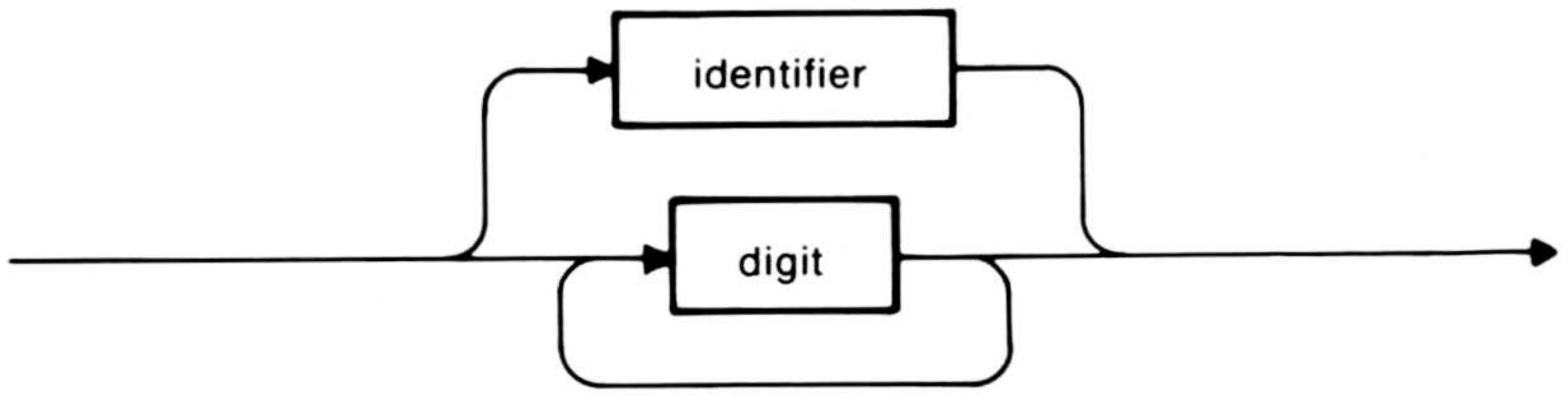
## Identifier



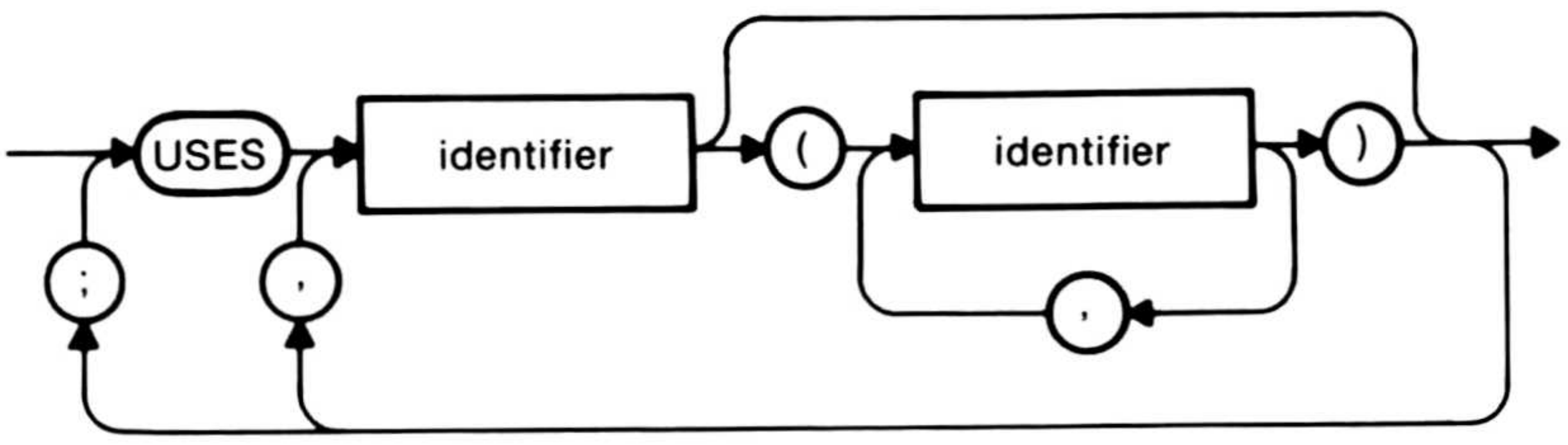
## Number



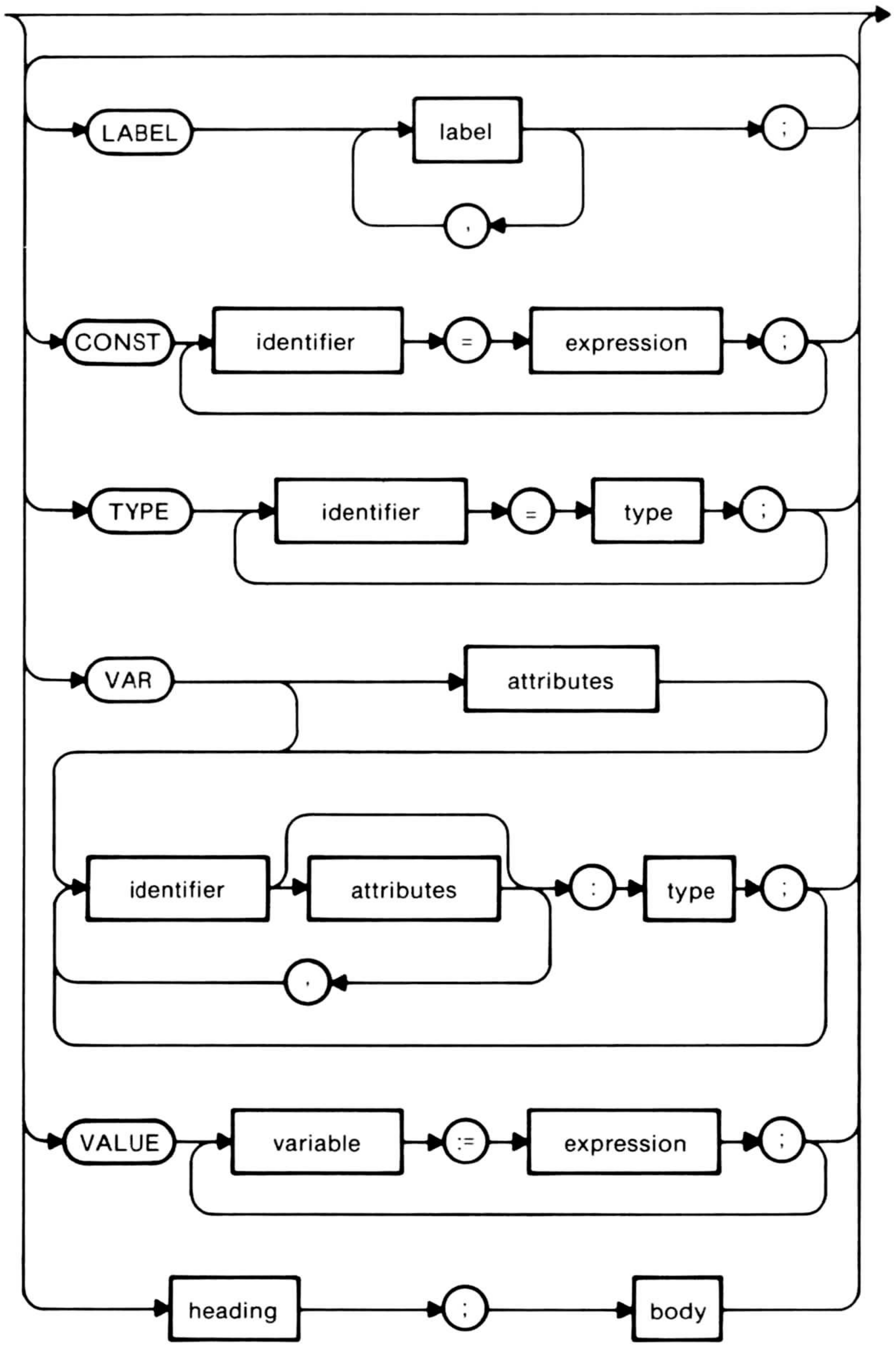
## Label



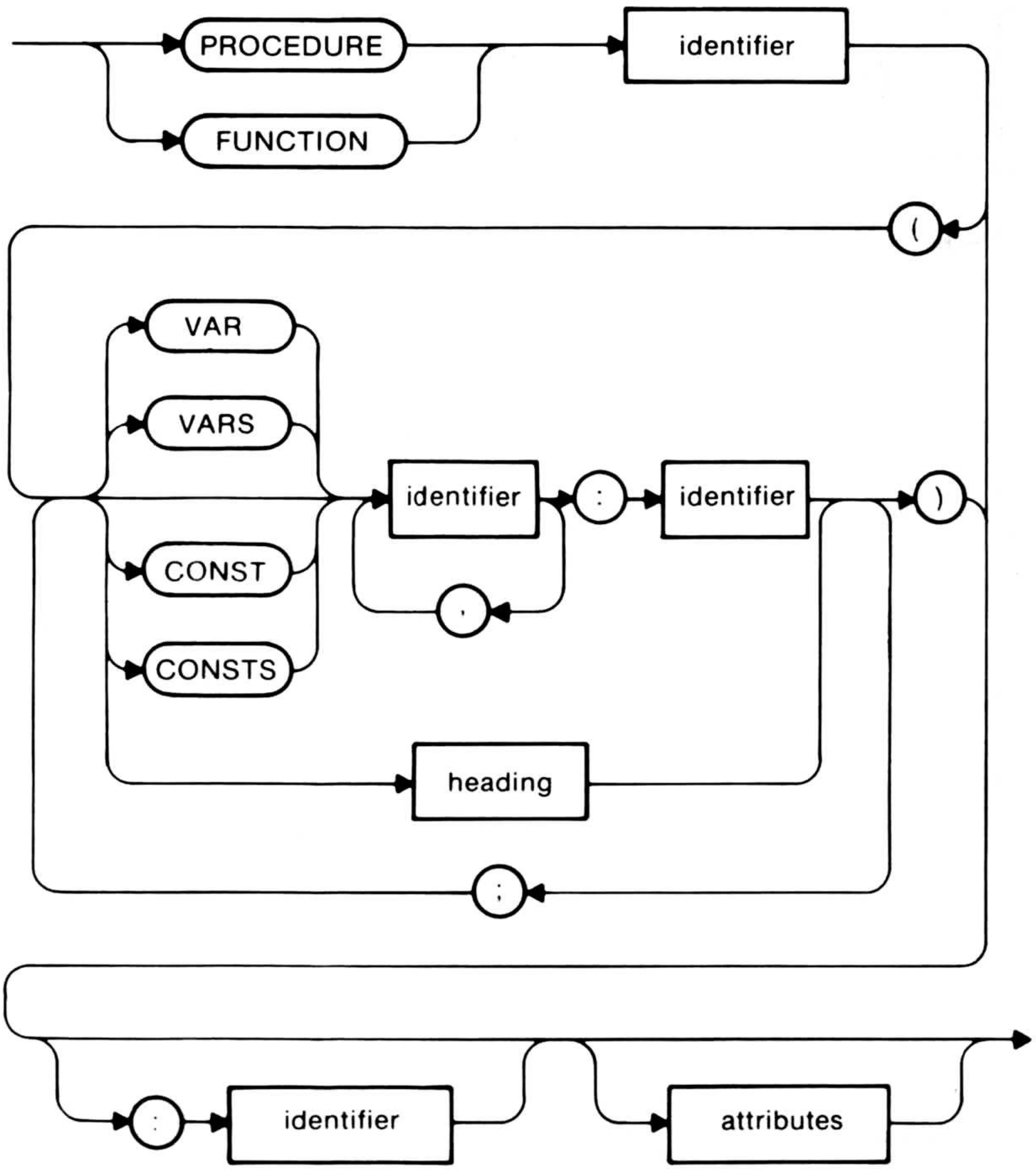
## Uselist



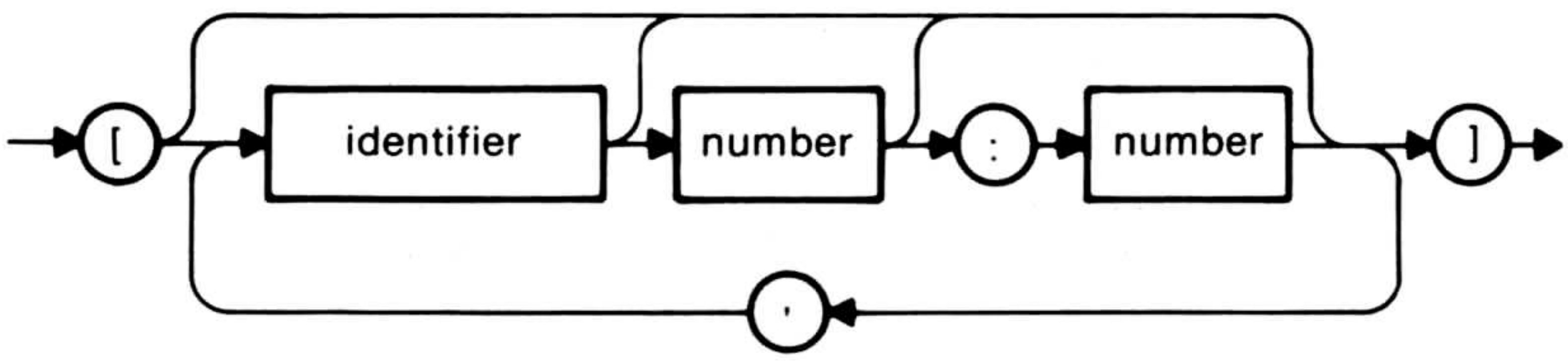
## Declarations



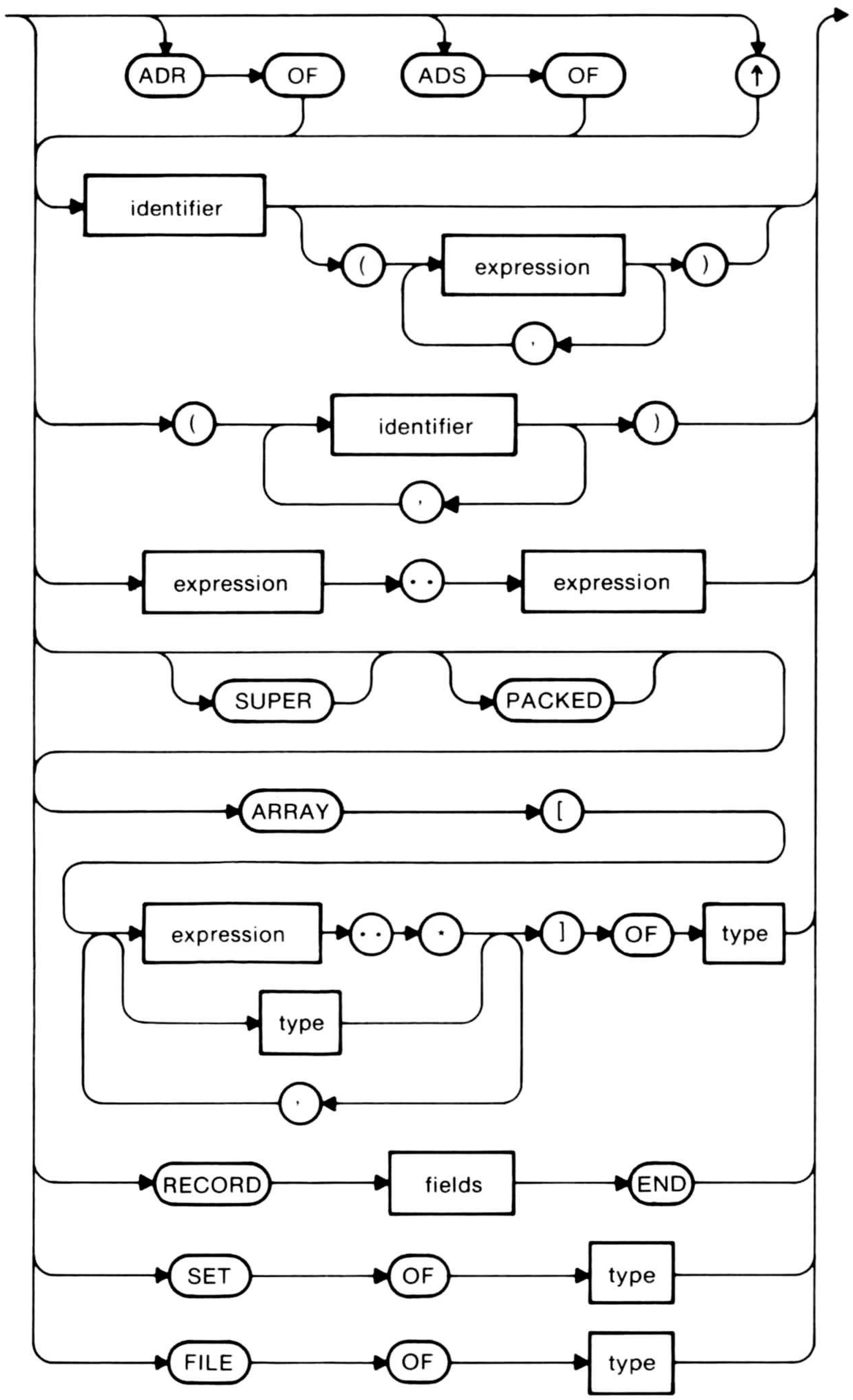
## Heading



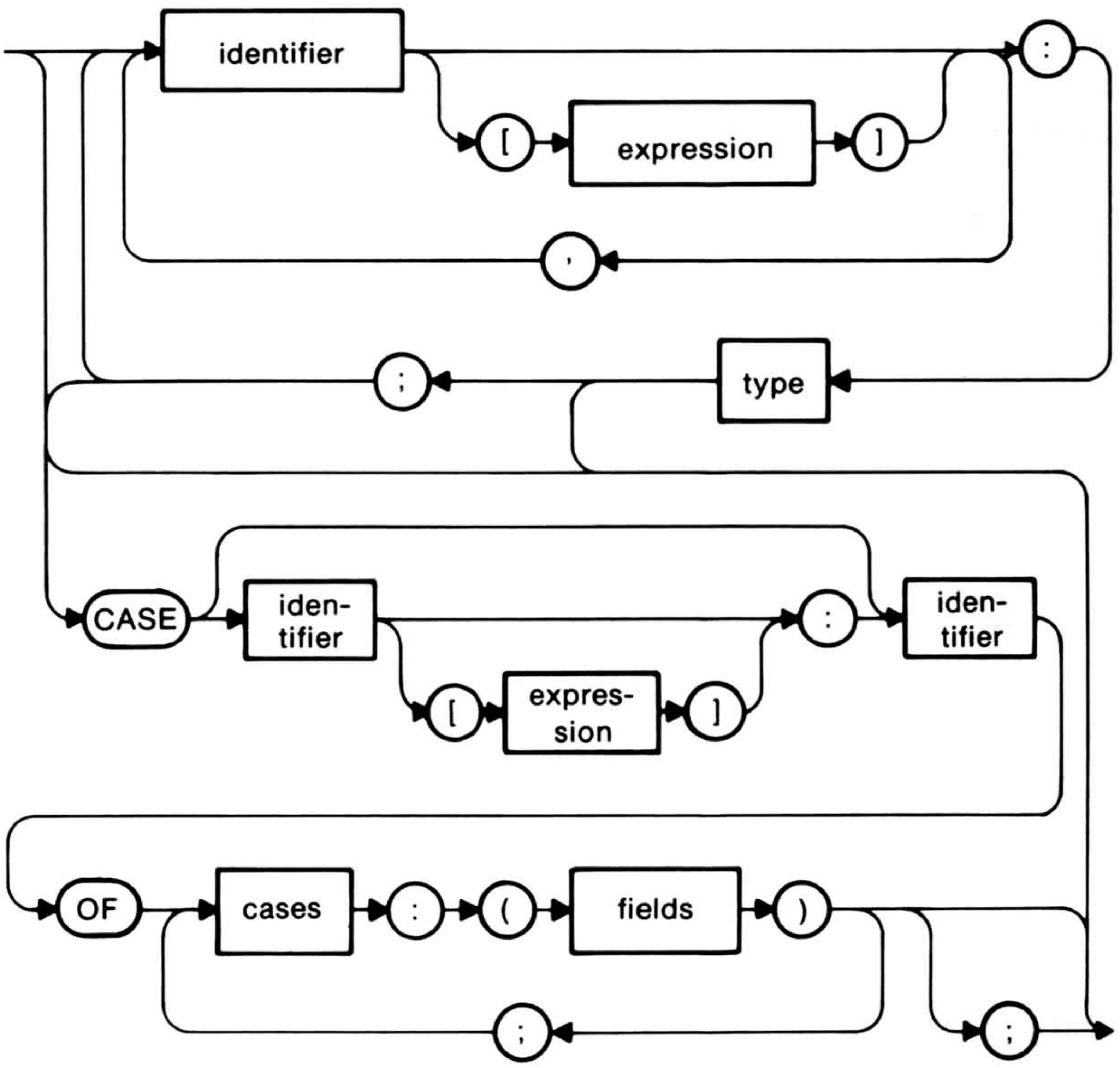
## Attributes



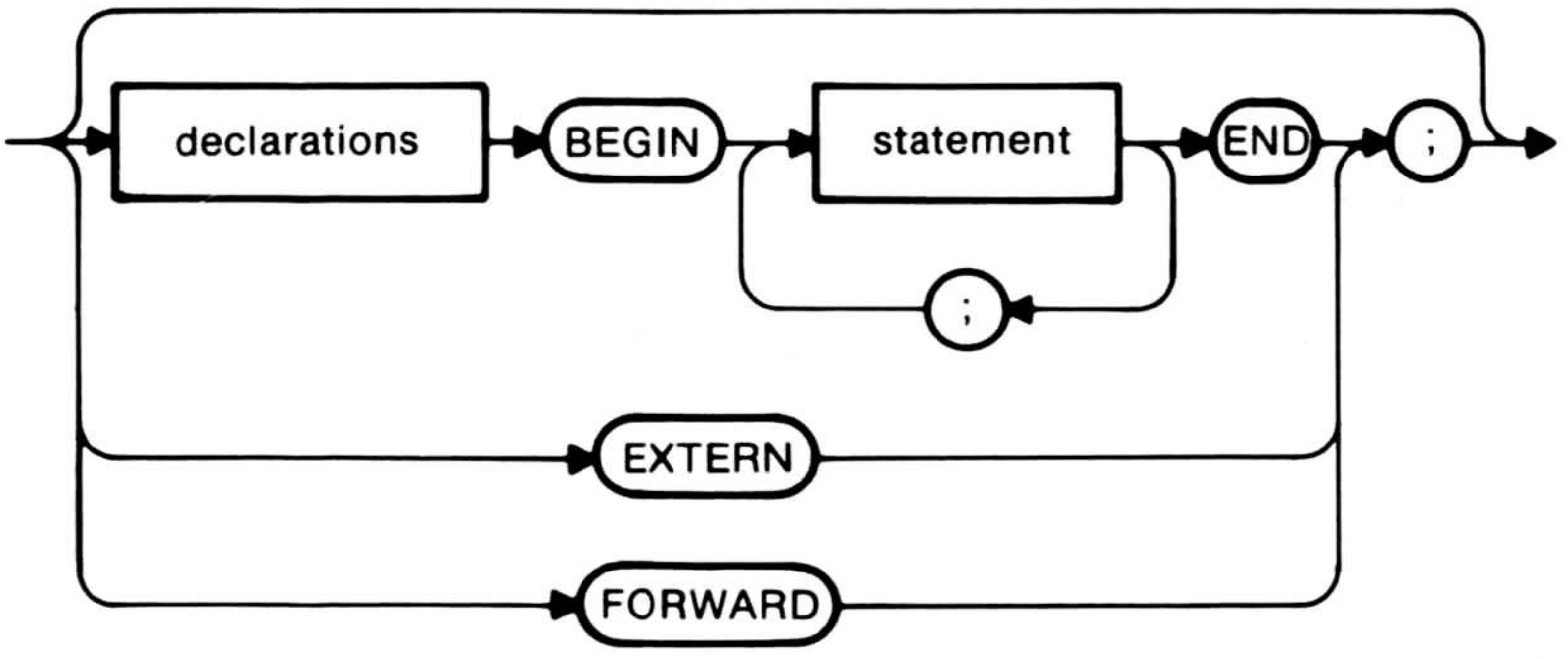
## Type


ADR
OF
ADS
OF
↑
identifier
(
expression
,
)
(
identifier
,
)
expression
..
expression
SUPER
PACKED
ARRAY
[
expression
..
*
type
,
]
OF
type
RECORD
fields
END
SET
OF
type
FILE
OF
type

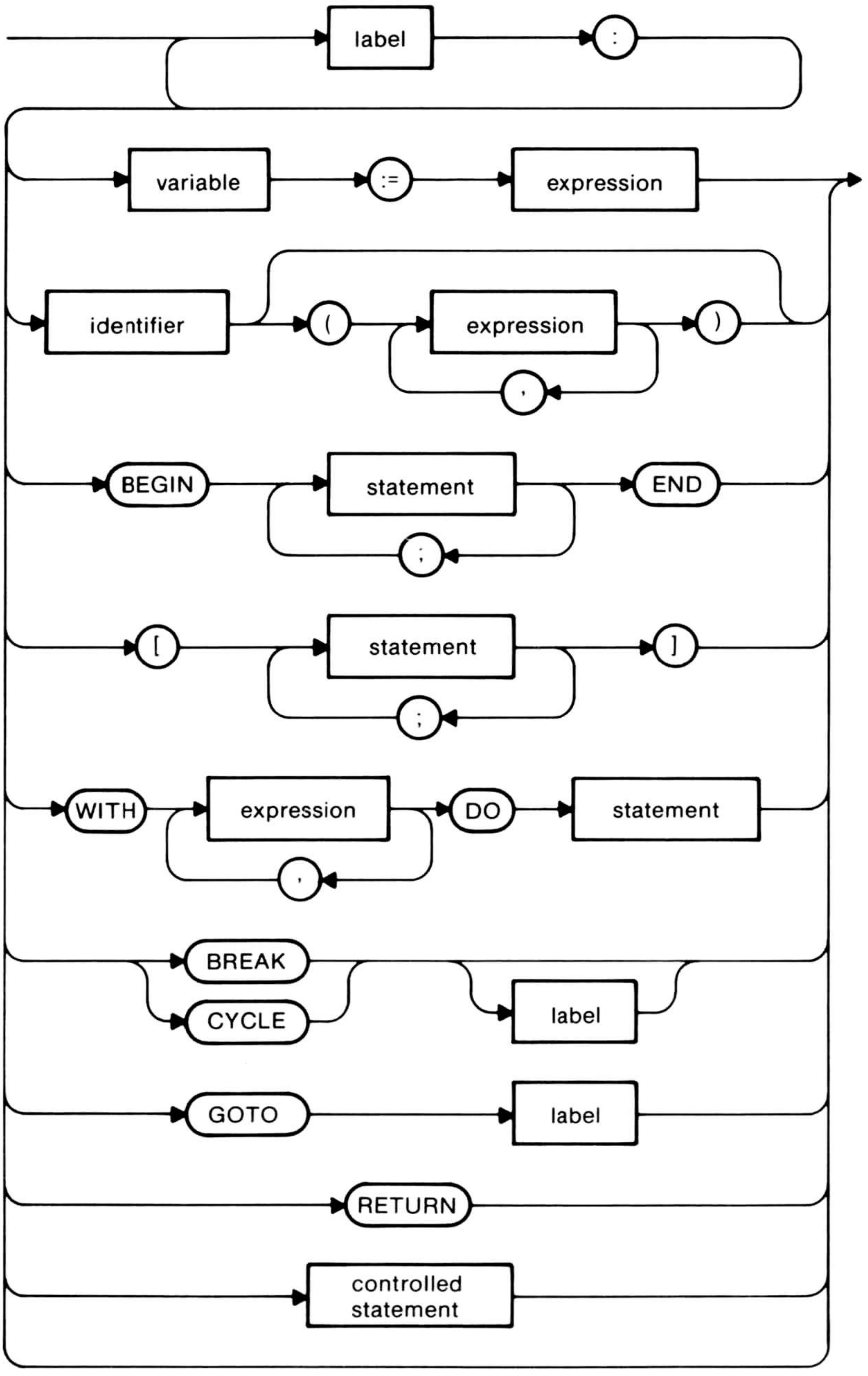
## Fields


identifier
[
expression
]
:
,
type
;
CASE
iden-tifier
[
expres-sion
]
:
iden-tifier
OF
cases
:
(
fields
)
;
;

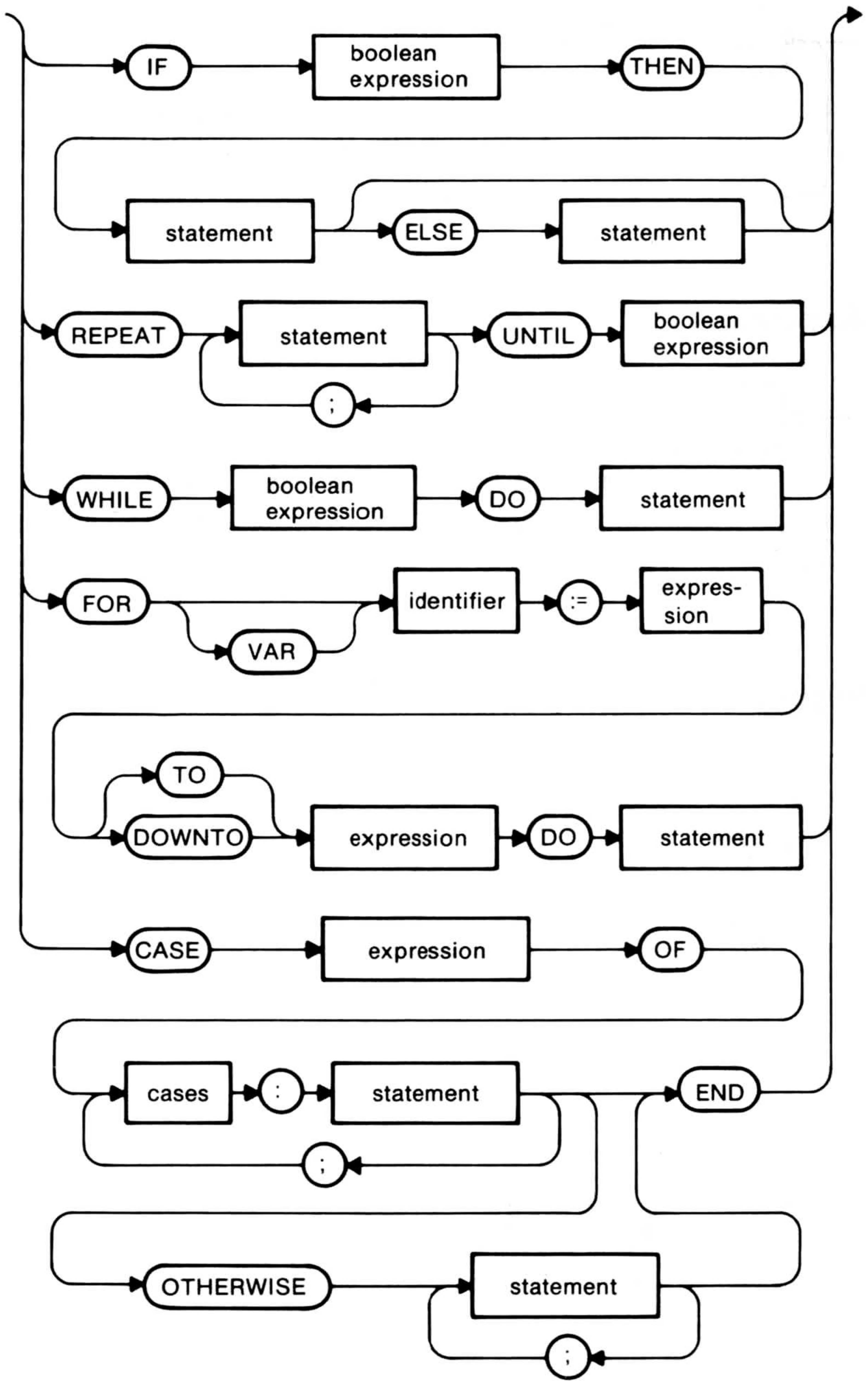
## Body


declarations
BEGIN
statement
END
;
;
EXTERN
FORWARD

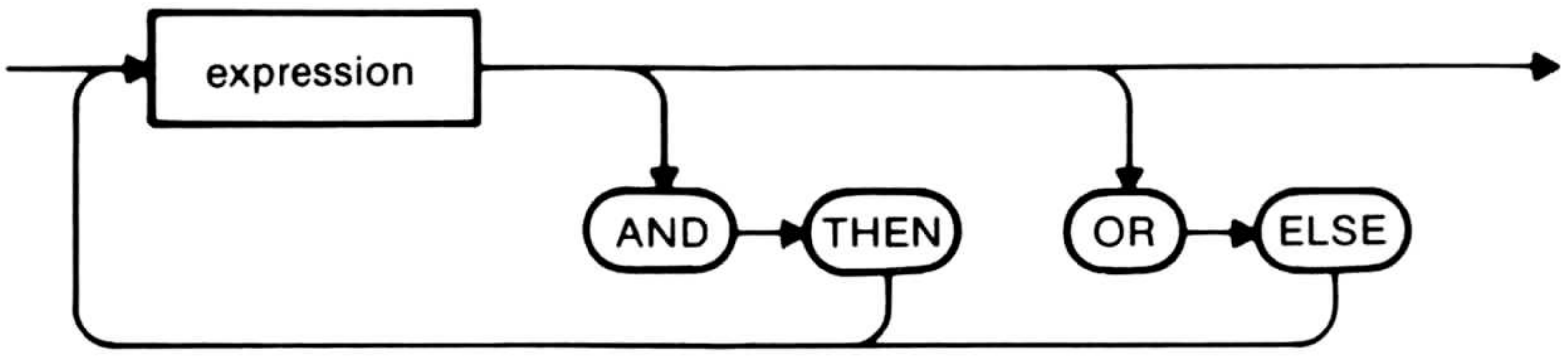
## Statement


label
:
variable
:=
expression
identifier
(
expression
,
)
BEGIN
statement
;
END
[
statement
;
]
WITH
expression
,
DO
statement
BREAK
CYCLE
label
GOTO
label
RETURN
controlled statement

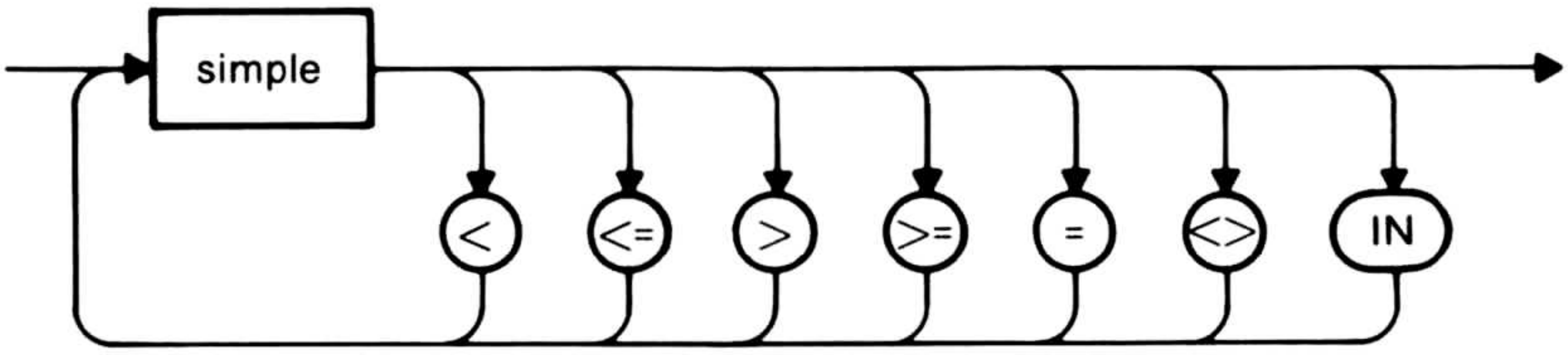
## Controlled Statement



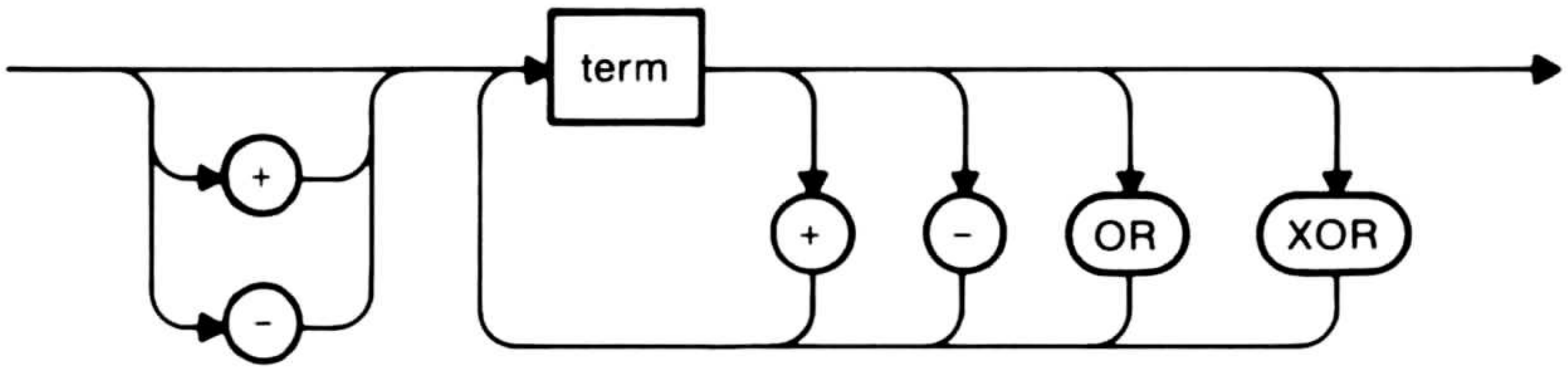
## Boolean Expression



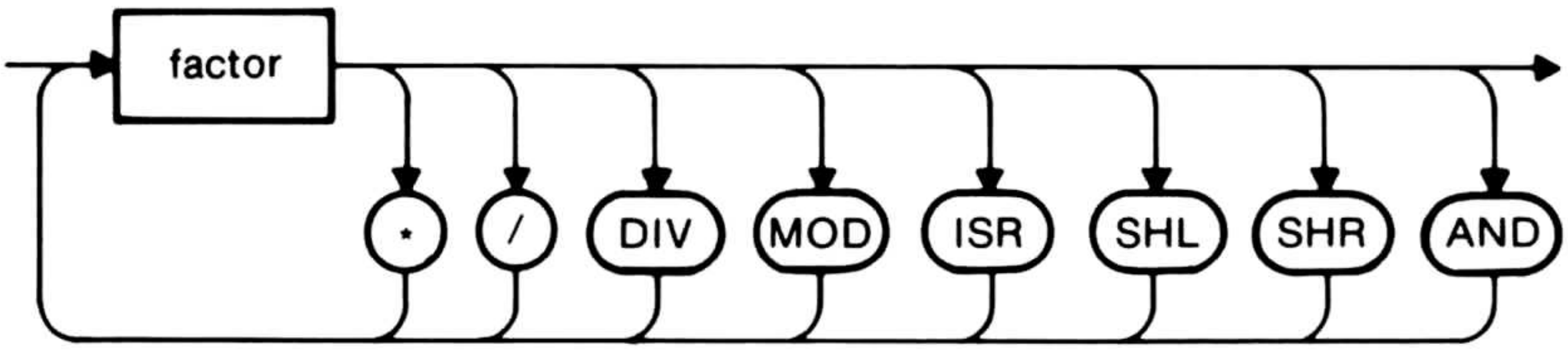
## Expression



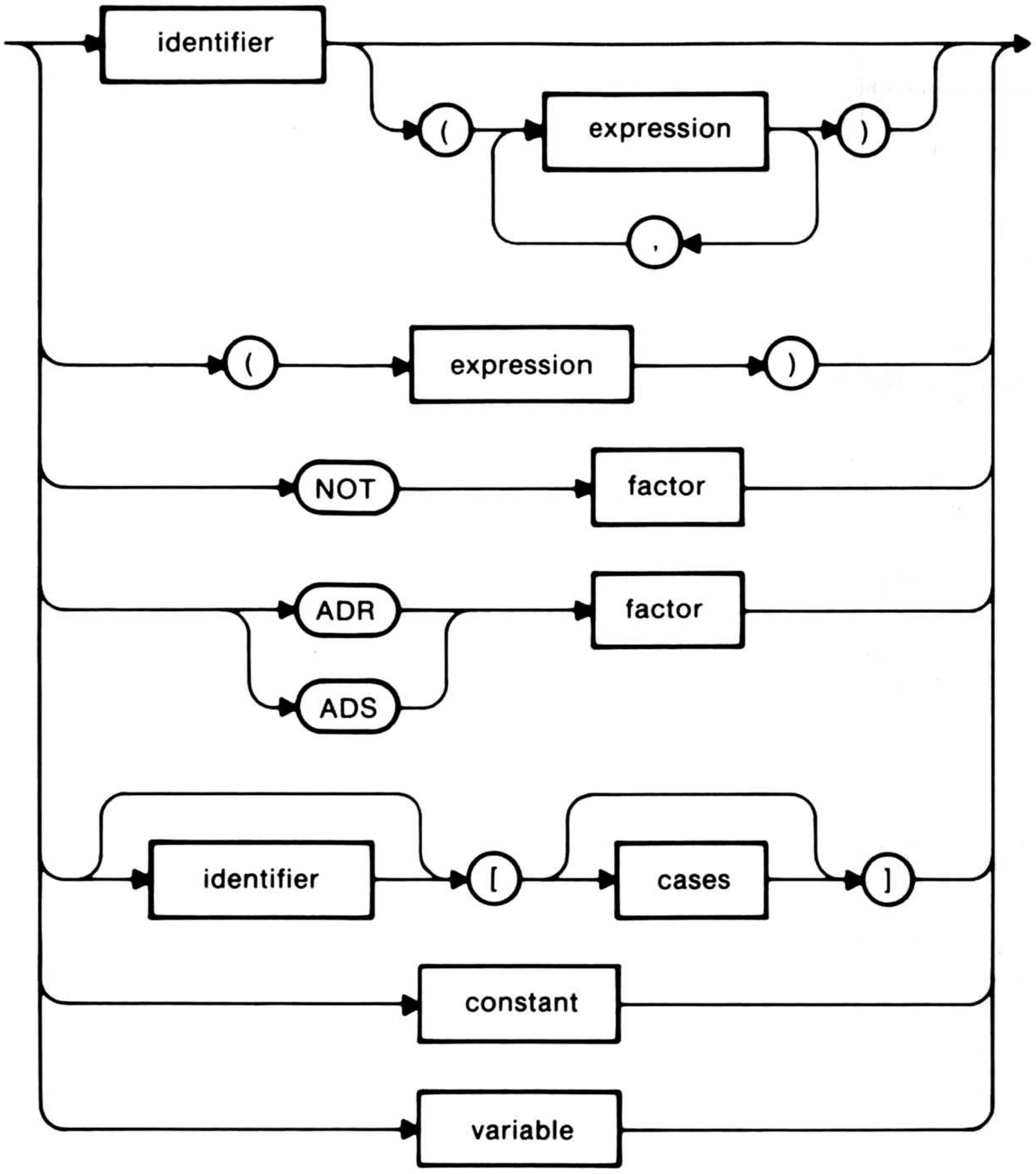
## Simple



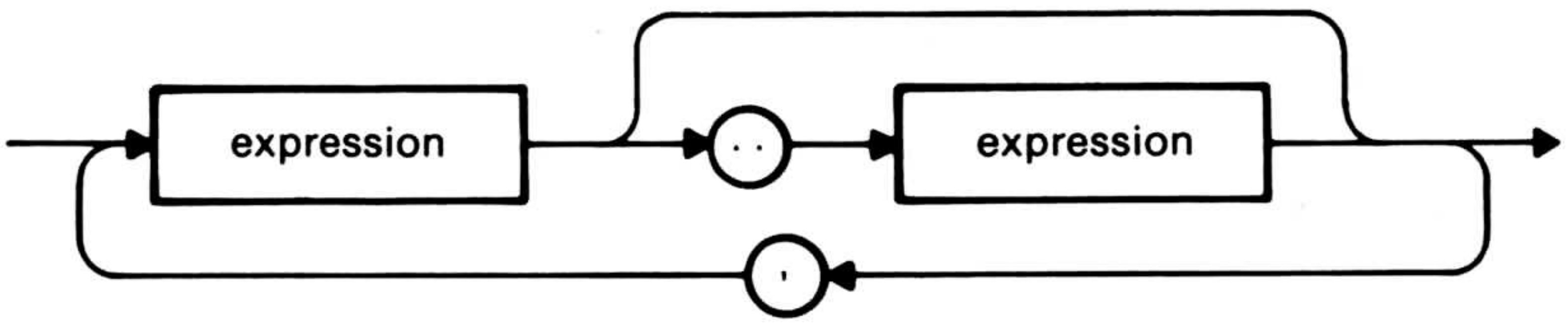
## Term



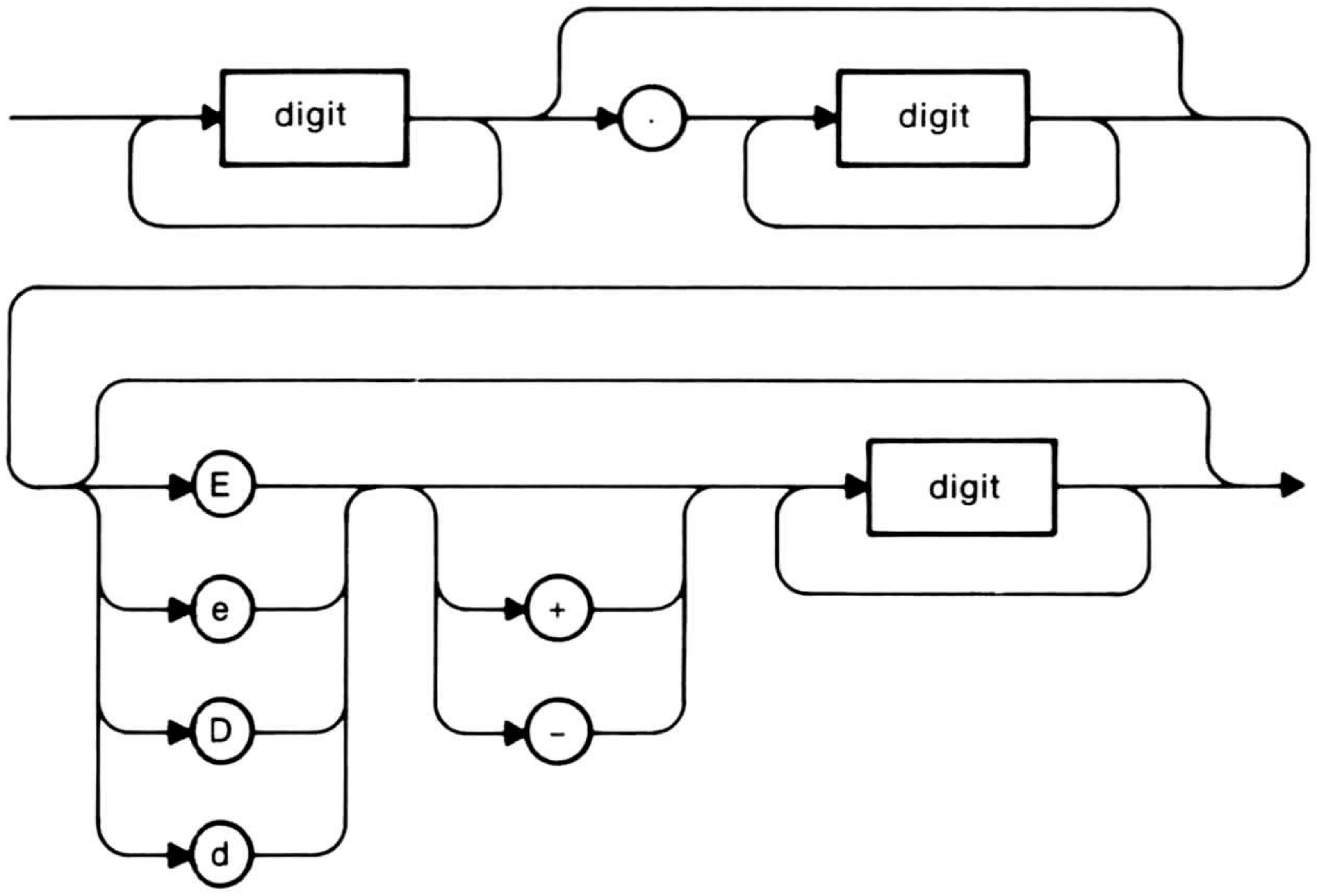
## Factor



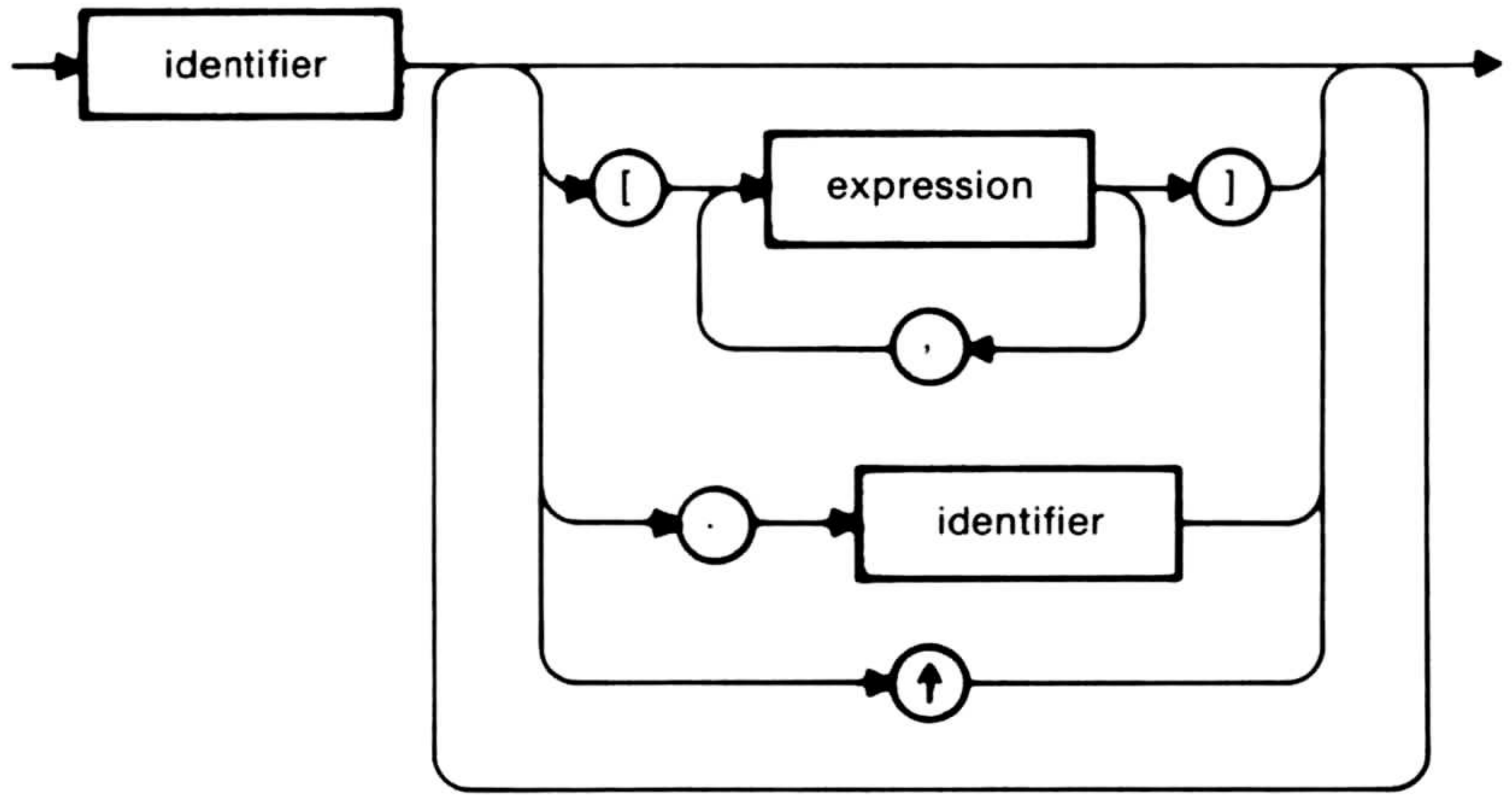
## Cases



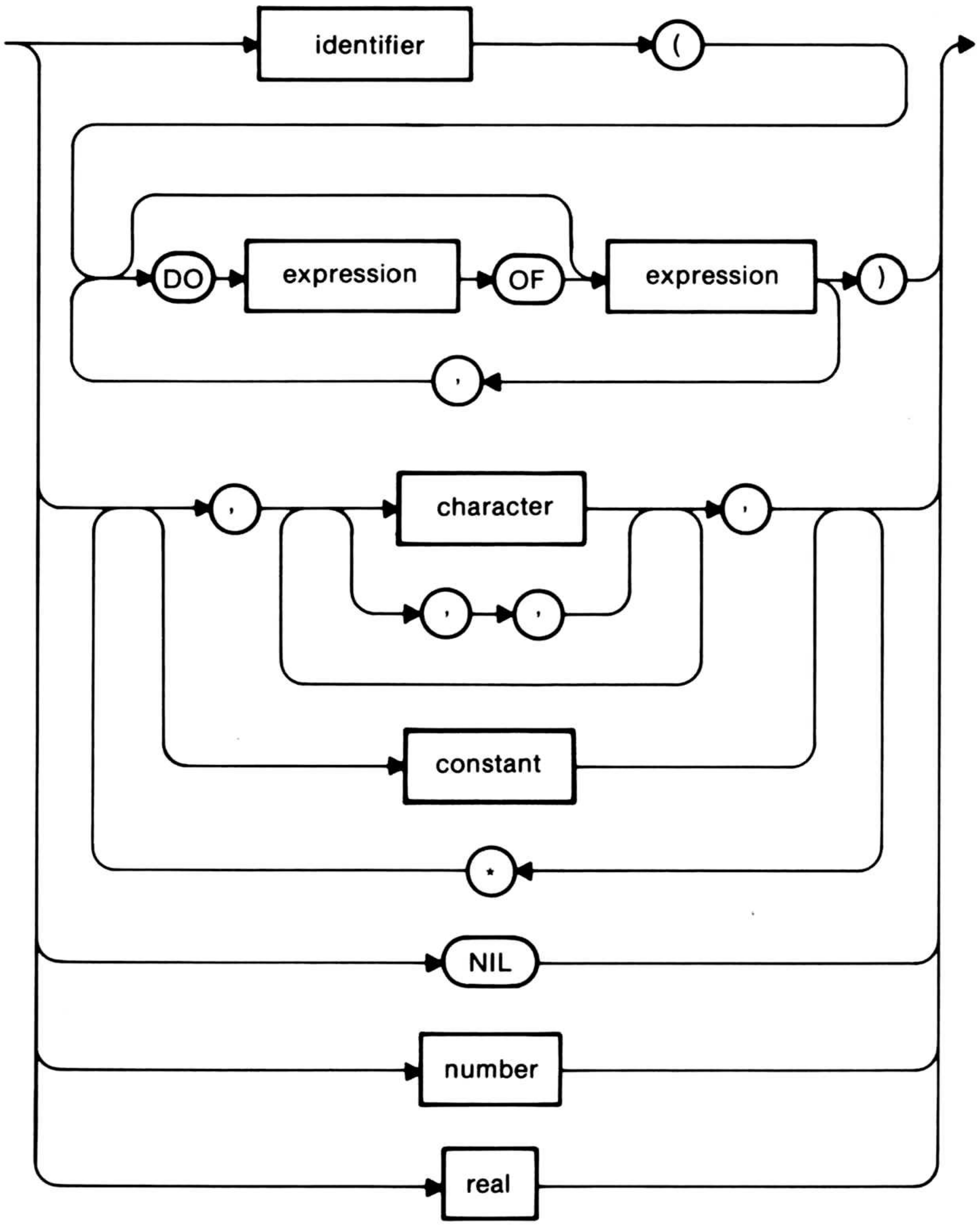
## Real



## Variable



## Constant


identifier
(
DO
expression
OF
expression
)
,
'
character
'
' '
constant
*
NIL
number
real

# Appendix B
# Microsoft Pascal Features and the ISO Standard

Microsoft Corporation fully supports the effort to standardize the Pascal language. A Microsoft representative sits on the ANSI/IEEE committee. At this writing, the ISO Pascal standard, Level 0 and Level 1, is still in draft status.

MS-Pascal generally conforms to this current draft standard, but does not yet implement the proposed conformant array mechanism. This controversial method of passing arrays of different bounds as one parameter type has not been tested, and the details change from draft to draft. The conformant array scheme is not part of the ANSI/IEEE standard nor the ISO Level 0 standard.

The super array type in MS-Pascal provides conformant array parameters, as well as dynamic length arrays allocated on the heap. Programs correctly written to the ISO standard (Level 0) or to the ANSI/IEEE standard should run correctly, without changes, under MS-Pascal. However, since MS-Pascal features introduce new reserved words and other elements, this goal cannot be fully realized.

## B.1 Microsoft Pascal and the ISO Standard

The ISO standard defines a large number of error conditions, but allows a particular implementation to handle an error by documenting the fact that the error is not caught. These "errors not caught," and other differences between MS-Pascal and the ISO standard, are described below. An MS-Pascal program that conforms or tests conformance to the ISO standard must have both the metacommands $standard and $debug on.

MS-Pascal allows the following minor extensions to the current ISO/ANSI/IEEE standard:

1. a question mark (?) as a substitute for the up arrow (^)
2. the underscore (_) in identifiers

Due to the way the compiler binds identifiers, the new reserved words added at the extend and system levels cannot be used as identifiers at the standard level. A new directive, EXTERN, and new predeclared functions are standard in MS-Pascal.

The current differences between Microsoft Pascal at the standard level and the current ISO/ANSI/IEEE standard are summarized in the following pages.

1. The ISO standard requires a separator between numbers and identifiers or keywords.

   In some cases, MS-Pascal doesn't require a separator between a number and an identifier or keyword, e.g., "100mod" is accepted as "100 mod" without error.

2. The ISO standard does not allow passing a component of a PACKED structure as a reference parameter.

   MS-Pascal specifically permits passing a CHAR element of a PACKED ARRAY [1 .. n] OF CHAR as a reference parameter. Passing a tag field as a reference is an error not caught. Passing other packed components gives the usual error.

3. The ISO standard does not include the textfile line-marker character in the set of CHAR values.

   MS-Pascal permits all 256 8-bit values as CHAR values; with some operating systems, a particular CHAR value (e.g., carriage return) is also the line marker character.

4. The ISO standard requires a variant to be given for all possible tag values.

   MS-Pascal permits a variant record declaration in which not all tag values are given.

5. The ISO standard requires that an identifier have only one meaning in any scope.

   In MS-Pascal, using an identifier and then redeclaring it in the same scope is an error not caught. For example, the following,

   ```
   CONST X = Y; VAR Y : CHAR;
   ```

   has two meanings for Y in the same scope. MS-Pascal generally uses the latest definition for an identifier. There is one ambiguous case: if you declare type FOO in one scope and in an inner scope TYPE P = ^ FOO; F00 = type; then FOO has two meanings and intent is ambiguous. In this case, the compiler uses the later definition of FOO and issues a warning.

6. The ISO standard requires field width "M" to be greater than zero in WRITE and WRITELN procedures.

   MS-Pascal treats M < 0 as if M = ABS (M), but field expansion takes place from the right rather than the left. M can also be zero, to WRITE nothing. Textfile READ (LN) and WRITE (LN) parameters can take both M and N parameters (ignored if not needed). The form "V::N" is allowed. When writing an INTEGER, the N parameter sets the output radix; when reading or writing an enumerated type, the N parameter sets the ordinal number or constant identifier option.

7. The ISO standard does not allow a variable created with the long form of NEW to be assigned, used in an expression, or passed as a parameter. However, this is difficult to check at compile time and expensive to check at runtime.

   MS-Pascal allows assignments to these variables using the actual length of the target variable. The ISO standard error is not caught.

8. The ISO standard does not allow the short form of DISPOSE to be used on a structure allocated with the long form of NEW. The ISO standard only permits a variable allocated with the long form of NEW to be released with the long form of DISPOSE, and all tag fields should never change between the calls.

   MS-Pascal allows the short form of DISPOSE to be used on a structure allocated with the long form of NEW, and does not check for changes in tag values.

9. The ISO standard declares that when a "change of variant" occurs (such as when a new tag value is assigned), all the variant fields become undefined.

   MS-Pascal does not set the fields uninitialized when a new tag is assigned and so does not catch use of a variant field with an undefined value.

10. The ISO standard does not allow a variable with an active reference (i.e., the records of an executing WITH statement or an actual reference parameter) to be disposed (if a heap variable) or changed by a GET or PUT (if a file buffer variable).

    MS-Pascal does not catch these as errors.

11. The ISO standard currently defines I MOD J as an error if $J < 0$ and the result of MOD is positive, even if I is negative.

    MS-Pascal does not currently use the new draft standard semantics for the MOD operator. Programs intended to be portable should not use MOD unless both operands are positive.

12. The ISO standard at Level 1 defines conformant array.

    MS-Pascal does not implement the conformant array concept in Level 1 of the ISO standard. Super arrays provide much the same functionality in a more flexible way.

13. The ISO standard requires the control variable of a FOR loop to be local to the immediate block. Any assignment to this control variable is an error.

    MS-Pascal allows a nonlocal variable to be used if it is STATIC, so either a local variable or one at the PROGRAM level can be a FOR statement control variable. MS-Pascal also does not detect an assignment to the control variable as an error if assignment occurs in a procedure or function called within the FOR statement.

14. The ISO standard requires the CHR argument to be INTEGER.

    MS-Pascal allows CHR to take any ordinal type.

## B.2 Summary of Microsoft Pascal Features

This outline summarizes MS-Pascal extensions to the ISO standard. Unless otherwise noted, all are at the extend level.

### B.2.1 Syntactic and Pragmatic Features

1. The metalanguage (standard level)

   $brave
   $debug
   $decmath
   $entry
   $errors
   $extend
   $floatcalls
   $goto
   $if $then $else $end
   $include
   $inconst
   $indexck
   $initck
   $integer
   $line
   $linesize
   $list
   $mathck
   $message
   $nilck
   $ocode
   $optbug
   $page
   $pageif
   $pagesize
   $pop
   $push
   $rangeck
   $real
   $rom
   $runtime
   $simple
   $size
   $skip
   $speed
   $stackck
   $standard
   $subtitle
   $symtab
   $system
   $stagck
   $title
   $warn

2. Extra listing (standard level)

   a. flags for jumps, globals, identifier level, control level, header, trailer

   b. textual error and warning messages

3. Syntactic additions
   a. ! as comment to end of line
   b. square brackets equivalent to BEGIN/END
4. Nondecimal number notation
   a. numeric constants with # or nn# (where nn = 2 . . 36)
   b. DECODE/READ takes # notation
   c. ENCODE/WRITE with N of 2, 8, 10, 16
5. Extended CASE range
   a. for CASE statements and record variants
   b. OTHERWISE for all other values
   c. A . . B for range of values

## B.2.2 Data Types and Modes

1. WORD type, WRD function, MAXWORD constant
2. REAL4 and REAL8 types
3. INTEGER4 type, MAXINT4 constant
4. FLOAT4, ROUND4, and TRUNC4 functions
5. Address types (system level)
   a. ADR and ADS types and operators
   b. VARS and CONSTS parameters
6. SUPER array types
   a. conformant parameters
   b. dynamic length heap variables
   c. multidimensional super arrays
   d. STRING and LSTRING super types
7. LSTRING type, NULL constant, .LEN field
8. Explicit byte offsets in records (system level)
9. CONST and CONSTS reference parameters for constants and expressions

10. Structured (array, record, and set) constants
11. Extended functions returning any assignable type
12. Variable selection on values returned from functions
13. Attributes

| | |
|---|---|
| EXTERN | PORT |
| EXTERNAL | PUBLIC |
| FORTRAN | PURE |
| INTERRUPT | READONLY |
| ORIGIN | STATIC |

## B.2.3 Operators and Intrinsics

1. Extend level operators:
   a. shift operators: SHL SHR ISR
   b. bitwise logical: AND OR NOT XOR
   c. set operators: < >
2. Constant expressions:
   a. string constant concatenation with * operator
   b. numeric, ordinal, Boolean expressions in type clauses
   c. other constant functions:

| | |
|---|---|
| CHR | UPPER |
| DIV | WRD |
| HIBYTE | * |
| HIWORD | + |
| LOBYTE | - |
| LOWER | < |
| LOWORD | <= |
| MOD | <> |
| ORD | = |
| RETYPE | > |
| SIZEOF | >= |

3. Additional intrinsic functions at extend level:

| | |
|---|---|
| ABORT | HIWORD |
| BYLONG | LOBYTE |
| BYWORD | LOWER |
| DECODE | LOWORD |
| ENCODE | RESULT |
| EVAL | SIZEOF |
| HIBYTE | UPPER |

4. Additional intrinsic functions at system level:

| | |
|---|---|
| FILLC | MOVESL |
| FILLSC | MOVESR |
| MOVEL | RETYPE |
| MOVER | |

5. Intrinsic functions that operate on strings:

   a. for STRING or LSTRING: COPYSTR POSITN SCANEQ SCANNE

   b. for LSTRING only: CONCAT INSERT DELETE COPYLST

6. MS-FORTRAN REAL library functions (standard level)

7. MS-Pascal library functions (standard level):

| | |
|---|---|
| ALLHQQ | MARKAS |
| BEGOQQ | MEMAVL |
| BEGXQQ | PLYUQQ |
| DATE | PTYUQQ |
| DISBIN | RELEAS |
| ENDOQQ | SADDOK |
| ENDXQQ | SMULOK |
| ENABIN | TICS |
| FREECT | TIME |
| GTYUQQ | UADDOK |
| LADDOK | UMULOK |
| LMULOK | UNLOCK |
| LOCKED | VECTIN |

## B.2.4 Control Flow and Structure Features

1. Control flow statements: BREAK, CYCLE, and RETURN

2. Sequential control operators: AND THEN and OR ELSE in IF, WHILE, REPEAT

3. Extended FOR loop: FOR VAR variable
4. VALUE section to initialize static variables
5. Mixed order LABEL, CONST, TYPE, VAR, VALUE sections
6. Compilable MODULES, with global attributes
7. UNIT INTERFACE and IMPLEMENTATION:
   a. interface version numbers, version checking
   b. optional rename of constituents
   c. guaranteed unique unit initialization
   d. optional unit initialization

## B.2.5 Extend Level I/O and Files

1. Textfile line length declaration, TEXT (nnn)
2. READ enumerated, Boolean, pointer, STRING, LSTRING
3. WRITE enumerated, pointer, LSTRING
4. Negative M value to justify left instead of right
5. Temporary files
6. DIRECT mode files, SEEK procedure
7. ASSIGN, CLOSE, DISCARD, READSET, READFN procedures
8. FILEMODES type and constants, F.MODE access
9. Error trapping, F.TRAP and F.ERRS access
10. Enumerated I/O using identifier as string

## B.2.6 System Level I/O

The MS-Pascal extension to the ISO standard offers full FCBFQQ type equivalent to FILE types.

# Appendix C
# Microsoft Pascal and Other Pascals

At the standard level, Microsoft Pascal conforms to the current ISO draft standard. In theory, therefore, programs written in accordance with the ISO standard are portable and can be compiled with any MS-Pascal compiler with no problem.

In practice, however, the majority of Pascal programs are written with at least some nonstandard features. In these cases, it is necessary to alter the Pascal source file to conform to the conventions used in Microsoft Pascal.

## C.1 Implementations of Pascal

The areas in which different implementations of the Pascal language differ from one another fall into one of the following categories:

1. Interactive I/O

   MS-Pascal implements lazy evaluation to handle interactive I/O in a natural way. Other Pascals may implement this feature in different ways. For example, some systems require an initial READLN.

2. String handling

   MS-Pascal supports the super array type LSTRING to handle variable length strings efficiently. The ISO standard provides the PACK and UNPACK procedures for dealing with strings; other Pascals often have some improvement on the string handling facilities described in the standard.

3. Compiler controls

   Compiler controls implemented either as command line switches or as commands within source comments vary from Pascal to Pascal. To ensure portability, eliminate all embedded controls from comments.

4. Maximum set size

   The maximum set size varies from Pascal to Pascal. Some Pascals limit set size to 16 or 64 elements. In MS-Pascal, sets may contain up to 256 elements. This allows support of the SET OF CHAR.

5. Type compatibility

   The rules for type compatibility vary in their strictness. In some Pascals, structurally equivalent types with different names are compatible; in others (and in the ISO standard), they are not.

6. Out of block GOTOs

   Some Pascals do not permit the out-of-block GOTOs that are permitted in MS-Pascal.

7. Heap management

   Rather than use the procedures NEW and DISPOSE for managing dynamic allocation of memory, some Pascals use the MARK and RELEASE procedures. MS-Pascal supports both methods. (MARKAS and RELEAS are the MS-Pascal names for MARK and RELEASE.)

8. OTHERWISE in CASE statements and variant records

   If OTHERWISE is omitted in a CASE statement, control does not automatically pass to the next executable statement as in some other extended Pascals. Also, some other Pascals use the word ELSE or OTHERS instead of OTHERWISE.

9. Assigning filenames

   The ASSIGN procedure in MS-Pascal sets an operating system filename for a file. Some other Pascals use a second parameter to RESET and REWRITE for the filename.

10. Separate compilation

    Most Pascals exclude the EXTERN (or EXTERNAL) directive for procedures and functions. Many support the idea of a MODULE and/or an INTERFACE and IMPLEMENTATION, although the syntax may differ. Some do not support PUBLIC and EXTERN variables, but may use a FORTRAN COMMON approach. In the latter case, for portability, you should give all global variables in one MS-Pascal VAR section, using [PUBLIC] in the PROGRAM and [EXTERN] in the MODULE, and $include the same variable declarations in each.

11. Program parameters

    Some Pascals ignore program parameters. In some Pascals, all files must be program parameters.

12. Procedural parameters

    Several Pascals do not permit passing procedures and functions as parameters. Many do not permit passing any predeclared procedures or functions.

## C.2 Microsoft Pascal and UCSD Pascal

Because UCSD Pascal® is one of the more prevalent Pascals for microcomputers, conversion of source files from UCSD to MS-Pascal, and vice versa, is likely to be a common occurrence. This section discusses the differences and similarities between the two Pascals.

MS-Pascal has incorporated many of the UCSD extensions in one form or another. Table C.1 compares UCSD extensions with similar extensions available in MS-Pascal.

**Table C.1**

**Microsoft Pascal and UCSD Pascal**

| UCSD Extension | MS-Pascal Equivalent |
|---|---|
| ATAN | ARCTAN |
| BLOCKREAD | GETUQQ |
| BLOCKWRITE | PUTUQQ |
| CLOSE | CLOSE |
| CLOSE (F, LOCK) | CLOSE (F) |
| CLOSE (F, PURGE) | DISCARD (F) |
| CONCAT | CONCAT |
| COPY | COPYLST or MOVEL |
| DELETE | DELETE |
| EXIT | RETURN or GOTO |
| FILLCHAR | FILLC and FILLSC |
| HALT | ENDXQQ |
| INSERT | INSERT |
| IORESULT, $I | ERRS and TRAP fields |
| LENGTH | .LEN or STR [0] |
| LOG | LNDRQQ |
| MARK | MARKAS |
| MEMAVAIL | MEMAVL |
| MOVELEFT | MOVEL and MOVESL |
| MOVERIGHT | MOVER and MOVESR |
| POS | POSITN |
| RELEASE | RELEAS |
| SCAN | SCANEQ and SCANNE |
| SEEK | SEEK |
| SIZEOF | SIZEOF |
| STR | ENCODE |
| STRING [n] | LSTRING (n) |
| UNIT | UNIT |
| Untyped Files | FCBFQQ type |

The following notes describe comparative points of interest.

1. The UCSD STRING [n] type is logically similar to the MS-Pascal LSTRING (n) type. Both contain the length of a variable length string in element zero of an ARRAY of CHAR.

2. UCSD Pascal allocates pointer variables on the heap with MARK and RELEASE. Other Pascals normally use NEW and DISPOSE. MS-Pascal permits both methods of dynamic memory allocation.

3. MS-Pascal units are like UCSD Pascal units, with the following exceptions. In MS-Pascal, an INTERFACE must appear first in any compiland using it. Since UCSD Pascal has its own special file system, the name of the unit can be used to find the interface filename in a standard way.

   MS-Pascal requires a list of all identifiers exported from the unit in the UNIT clause itself and makes it optional in a USES clause. Different identifiers may be given in a USES clause to avoid identifier conflicts.

   Finally, MS-Pascal provides for unit initialization code and interface version control. Neither of these are available in UCSD Pascal.

4. CONCAT is a function in UCSD Pascal; in MS-Pascal, it is a procedure.

5. In UCSD Pascal, when a CASE statement whose control value does not select a statement is executed, the statement following the CASE statement is executed. In MS-Pascal, you must include an empty OTHERWISE clause to obtain this effect.

6. UCSD Pascal permits the use of the EOF (F) and EOLN (F) functions on a closed file; in MS-Pascal, this is an error.

7. UCSD Pascal permits comparison of records and arrays with the equal sign (=) and the not-equal sign (<>). In MS-Pascal, you must RETYPE the records and arrays to the same length STRING type, and then compare them as strings.

# Appendix D ASCII Character Codes

Table D.1

ASCII Character Codes

| Dec | Hex | CHR | Dec | Hex | CHR |
|---|---|---|---|---|---|
| 000 | 00H | NUL | 032 | 20H | SPACE |
| 001 | 01H | SOH | 033 | 21H | ! |
| 002 | 02H | STX | 034 | 22H | " |
| 003 | 03H | ETX | 035 | 23H | # |
| 004 | 04H | EOT | 036 | 24H | $ |
| 005 | 05H | ENQ | 037 | 25H | % |
| 006 | 06H | ACK | 038 | 26H | & |
| 007 | 07H | BEL | 039 | 27H | ' |
| 008 | 08H | BS | 040 | 28H | ( |
| 009 | 09H | HT | 041 | 29H | ) |
| 010 | 0AH | LF | 042 | 2AH | * |
| 011 | 0BH | VT | 043 | 2BH | + |
| 012 | 0CH | FF | 044 | 2CH | , |
| 013 | 0DH | CR | 045 | 2DH | - |
| 014 | 0EH | SO | 046 | 2EH | . |
| 015 | 0FH | SI | 047 | 2FH | / |
| 016 | 10H | DLE | 048 | 30H | 0 |
| 017 | 11H | DC1 | 049 | 31H | 1 |
| 018 | 12H | DC2 | 050 | 32H | 2 |
| 019 | 13H | DC3 | 051 | 33H | 3 |
| 020 | 14H | DC4 | 052 | 34H | 4 |
| 021 | 15H | NAK | 053 | 35H | 5 |
| 022 | 16H | SYN | 054 | 36H | 6 |
| 023 | 17H | ETB | 055 | 37H | 7 |
| 024 | 18H | CAN | 056 | 38H | 8 |
| 025 | 19H | EM | 057 | 39H | 9 |
| 026 | 1AH | SUB | 058 | 3AH | : |
| 027 | 1BH | ESCAPE | 059 | 3BH | ; |
| 028 | 1CH | FS | 060 | 3CH | < |
| 029 | 1DH | GS | 061 | 3DH | = |
| 030 | 1EH | RS | 062 | 3EH | > |
| 031 | 1FH | US | 063 | 3FH | ? |

**Table D.1** *(continued)*

| Dec | Hex | CHR | Dec | Hex | CHR |
|---|---|---|---|---|---|
| 064 | 40H | @ | 096 | 60H | ' |
| 065 | 41H | A | 097 | 61H | a |
| 066 | 42H | B | 098 | 62H | b |
| 067 | 43H | C | 099 | 63H | c |
| 068 | 44H | D | 100 | 64H | d |
| 069 | 45H | E | 101 | 65H | e |
| 070 | 46H | F | 102 | 66H | f |
| 071 | 47H | G | 103 | 67H | g |
| 072 | 48H | H | 104 | 68H | h |
| 073 | 49H | I | 105 | 69H | i |
| 074 | 4AH | J | 106 | 6AH | j |
| 075 | 4BH | K | 107 | 6BH | k |
| 076 | 4CH | L | 108 | 6CH | l |
| 077 | 4DH | M | 109 | 6DH | m |
| 078 | 4EH | N | 110 | 6EH | n |
| 079 | 4FH | O | 111 | 6FH | o |
| 080 | 50H | P | 112 | 70H | p |
| 081 | 51H | Q | 113 | 71H | q |
| 082 | 52H | R | 114 | 72H | r |
| 083 | 53H | S | 115 | 73H | s |
| 084 | 54H | T | 116 | 74H | t |
| 085 | 55H | U | 117 | 75H | u |
| 086 | 56H | V | 118 | 76H | v |
| 087 | 57H | W | 119 | 77H | w |
| 088 | 58H | X | 120 | 78H | x |
| 089 | 59H | Y | 121 | 79H | y |
| 090 | 5AH | Z | 122 | 7AH | z |
| 091 | 5BH | [ | 123 | 7BH | { |
| 092 | 5CH | \ | 124 | 7CH | \| |
| 093 | 5DH | ] | 125 | 7DH | } |
| 094 | 5EH | ^ | 126 | 7EH | ~ |
| 095 | 5FH | _ | 127 | 7FH | DEL |

Dec=decimal, Hex=hexadecimal (H), CHR=character, LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rub out

# Appendix E
# Summary of Microsoft Pascal Reserved Words

Reserved words at the standard level:

| | |
|---|---|
| AND | NIL |
| ARRAY | NOT |
| BEGIN | OF |
| CASE | OR |
| CONST | PACKED |
| DIV | PROCEDURE |
| DO | PROGRAM |
| DOWNTO | RECORD |
| ELSE | REPEAT |
| END | SET |
| FILE | THEN |
| FOR | TO |
| FUNCTION | TYPE |
| GOTO | UNTIL |
| IF | VAR |
| IN | WHILE |
| LABEL | WITH |
| MOD | |

Additional reserved words at the extend level:

| | |
|---|---|
| BREAK | RETURN |
| CONSTS | SHL |
| CYCLE | SHR |
| IMPLEMENTATION | UNIT |
| INTERFACE | USES |
| ISR | VALUE |
| MODULE | VARS |
| OTHERWISE | XOR |

Additional reserved words at the system level:

ADR
ADS

Names of attributes:

| | |
|---|---|
| EXTERN | PORT |
| EXTERNAL | PUBLIC |
| FORTRAN | PURE |
| INTERRUPT | READONLY |
| ORIGIN | STATIC |

Names of directives:

EXTERN
EXTERNAL
FORWARD

Logically, directives are reserved words. Since additional directives are allowed in ISO Pascal, all are included at the standard level. Note that EXTERN is both a directive and an attribute; EXTERNAL is a synonym for EXTERN in both cases. This provides compatibility with a number of other Pascals.

# Appendix F
# Summary of Available Procedures and Functions

This appendix provides a summary listing of all available functions and procedures, along with the name of the group in which they are presented in Section 15.1, "Categories of Available Procedures and Functions."

**Table F.1**

**Procedures and Functions**

| Name | Description | Category |
|---|---|---|
| ABORT | Terminate program | Extend level |
| ABS | Absolute value function | Arithmetic |
| ACDRQQ | REAL8 arc cosine function | Arithmetic |
| ACSRQQ | REAL4 arc cosine function | Arithmetic |
| AIDRQQ | REAL8 truncate function | Arithmetic |
| AISRQQ | REAL4 truncate function | Arithmetic |
| ALLHQQ | Allocate heap item | Library |
| ALLMQQ | Returns a long segmented address of type ADSMEM | Library |
| ANDRQQ | REAL8 round away from zero | Arithmetic |
| ANSRQQ | REAL4 round away from zero | Arithmetic |
| ARCTAN | Arc tangent function | Arithmetic |
| ASDRQQ | REAL8 arc sine function | Arithmetic |
| ASSRQQ | REAL4 arc sine function | Arithmetic |
| ASSIGN | Assign filename | File system |
| ATDRQQ | REAL8 arc tangent function | Arithmetic |
| ATSRQQ | REAL4 arc tangent function | Arithmetic |
| A2DRQQ | REAL8 arc tangent (A/B) | Arithmetic |
| A2SRQQ | REAL4 arc tangent (A/B) | Arithmetic |
| BEGOQQ | Initialize user | Library |
| BEGXQQ | Overall initialization | Library |
| BYLONG | WORD or INTEGER to INTEGER4 | Extend level |
| BYWORD | Put bytes in word | Extend level |

**Table F.1** *(continued)*

| Name | Description | Category |
|---|---|---|
| CHDRQQ | REAL8 hyperbolic cosine | Arithmetic |
| CHR | Get ASCII char of value | Data conversion |
| CHSRQQ | REAL4 hyperbolic cosine | Arithmetic |
| CLOSE | Close file | File system |
| CNDRQQ | REAL8 cosine function | Arithmetic |
| CNSRQQ | REAL4 cosine function | Arithmetic |
| CONCAT | Concatenate LSTRING | String |
| COPYLST | Copy to LSTRING | String |
| COPYSTR | Copy to STRING | String |
| COS | Cosine function | Arithmetic |
| DATE | Date function | Library |
| DECODE | Decode LSTRING to variable | Extend level |
| DELETE | Remove portion of LSTRING | String |
| DISBIN | Disable interrupts | Library |
| DISCARD | Close and delete file | File system |
| DISMQQ | Returns space to long heap memory pool | Library |
| DISPOSE | Dispose of heap item | Dynamic alloc'n |
| ENABIN | Enable interrupts | Library |
| ENCODE | Encode expression to LSTRING | Extend level |
| ENDOQQ | User termination | Library |
| ENDXQQ | Program termination | Library |
| EOF | Boolean end-of-file | File system |
| EOLN | Boolean end-of-line | File system |
| EVAL | Evaluate functions | Extend level |
| EXDRQQ | REAL8 exponential function | Arithmetic |
| EXP | Exponential function | Arithmetic |
| EXSRQQ | REAL4 exponential function | Arithmetic |
| FILLC | Fill area with C, relative | System level |
| FILLSC | Fill area with C, segmented | System level |
| FLOAT | Convert INTEGER to REAL | Data conversion |
| FLOAT4 | Convert INTEGER4 to REAL | Data conversion |
| FREECT | Give count of free blocks | Library |
| FREMQQ | Returns space to long heap memory pool | Library |
| GET | Get next file component | File system |
| GETMQQ | Returns long segmented address of type ADSMEM | Library |
| GTYUQQ | Direct terminal input | Library |
| HIBYTE | Get high BYTE | Extend level |
| HIWORD | Get high WORD | Extend level |

**Table F.1** *(continued)*

| Name | Description | Category |
|---|---|---|
| INSERT | Insert string | String |
| LADDOK | 32-bit signed addition check | Library |
| LDDRQQ | REAL8 log base ten function | Arithmetic |
| LDSRQQ | REAL4 log base ten function | Arithmetic |
| LMULOK | 32-bit signed multiply check | Library |
| LN | Natural log function | Arithmetic |
| LNDRQQ | REAL8 natural log | Arithmetic |
| LNSRQQ | REAL4 natural log | Arithmetic |
| LOBYTE | Get low BYTE | Extend level |
| LOCKED | Resource locked status | Library |
| LOWER | Get lower bound | Extend level |
| LOWORD | Get low WORD | Extend level |
| MARKAS | Mark heap bounds | Library |
| MEMAVL | Available memory | Library |
| MDDRQQ | REAL8 modulo function | Arithmetic |
| MDSRQQ | REAL4 modulo function | Arithmetic |
| MNDRQQ | REAL8 minimum function | Arithmetic |
| MNSRQQ | REAL4 minimum function | Arithmetic |
| MOVEL | Move bytes left, relative | System level |
| MOVER | Move bytes right, relative | System level |
| MOVESL | Move bytes left, segmented | System level |
| MOVESR | Move bytes right, segmented | System level |
| MXDRQQ | REAL8 maximum function | Arithmetic |
| MXSRQQ | REAL4 maximum function | Arithmetic |
| NEW | Allocate new heap item | Dynamic alloc'n |
| ODD | Boolean odd function | Data conversion |
| ORD | Get ordinal value | Data conversion |
| PACK | Pack CHAR array | Data conversion |
| PAGE | Write new page | File system |
| PIDRQQ | REAL8 to INTEGER power | Arithmetic |
| PISRQQ | REAL4 to INTEGER power | Arithmetic |
| PLYUQQ | Direct terminal end line | Library |
| POSITN | Find position of substring | String |
| PRED | Predecessor function | Data conversion |
| PRDRQQ | REAL8 to REAL8 power | Arithmetic |
| PRSRQQ | REAL4 to REAL4 power | Arithmetic |
| PTYUQQ | Direct terminal output | Library |
| PUT | Put value to file | File system |

**Table F.1** *(continued)*

| Name | Description | Category |
|---|---|---|
| READ | Read file | File system |
| READFN | Read filename | File system |
| READLN | Read file to end of line | File system |
| READSET | Read set | File system |
| RELEAS | Release heap space | Library |
| RESET | Ready file for read | File system |
| RESULT | Return result of function | Extend level |
| RETYPE | Force expression to type | System level |
| REWRITE | Ready file for write | File system |
| ROUND | Round REAL to INTEGER | Data conversion |
| ROUND4 | Round REAL to INTEGER4 | Data conversion |
| SADDOK | 16-bit signed addition check | Library |
| SCANEQ | Scan until char found | String |
| SCANNE | Scan until char not found | String |
| SEEK | Position at direct file record | File system |
| SHDRQQ | REAL8 hyperbolic sine | Arithmetic |
| SHSRQQ | REAL4 hyperbolic sine | Arithmetic |
| SIN | Sine function | Arithmetic |
| SIZEOF | Get size of structure | Extend level |
| SMULOK | 16-bit signed multiply check | Library |
| SNDRQQ | REAL8 sine function | Arithmetic |
| SNSRQQ | REAL4 sine function | Arithmetic |
| SQR | Square function | Arithmetic |
| SQRT | Square root function | Arithmetic |
| SRDRQQ | REAL8 square root | Arithmetic |
| SRSRQQ | REAL4 square root | Arithmetic |
| SUCC | Successor function | Data conversion |
| THDRQQ | REAL8 hyperbolic tangent | Arithmetic |
| THSRQQ | REAL4 hyperbolic tangent | Arithmetic |
| TICS | Time in arbitrary units | Library |
| TIME | Time of day function | Library |
| TNDRQQ | REAL8 tangent function | Arithmetic |
| TNSRQQ | REAL4 tangent function | Arithmetic |
| TRUNC | Truncate REAL to INTEGER | Data conversion |
| TRUNC4 | Truncate REAL to INTEGER4 | Data conversion |

**Table F.1** *(continued)*

| Name | Description | Category |
|---|---|---|
| UADDOK | Unsigned addition check | Library |
| UMULOK | Unsigned multiply check | Library |
| UNLOCK | Unlock resource | Library |
| UNPACK | Unpack STRING to array | Data conversion |
| UPPER | Get upper bound | Extend level |
| VECTIN | Set interrupt vector | Library |
| WRD | Convert to WORD value | Data conversion |
| WRITE | Write file | File system |
| WRITELN | Write line to file | File system |

# Appendix G
# Summary of Microsoft Pascal Metacommands

This appendix provides a single alphabetical list of all of the metacommands described in Chapter 18, "Microsoft Pascal Metacommands." Defaults, if any, are shown following the metacommand.

**Table G.1**

**Microsoft Pascal Metacommands**

| Metacommand | Action |
|---|---|
| $brave+ | Sends messages to the terminal screen |
| $debug- | Turns on or off all error checking (CK) |
| $decmath- | Directs the compiler to carry out floating-point operations using the decimal support routines in DECMATH.LIB |
| $entry- | Generates procedure entry and exit calls for debugger |
| $errors:25 | Sets number of errors allowed per page |
| $extend | Adds extend level features |
| $floatcalls+ | Directs the compiler to generate calls to real number floating-point routines |
| $goto- | Flags GOTOs as "considered harmful" |
| $if *constant*<br>  $then *text1*<br>  $else *text2*<br>$end | Allows conditional compilation of *text1* source if *constant* is greater than zero |
| $include:'*file*' | Switches compilation to file named |
| $inconst | Allows interactive setting of constant values at compile time |
| $indexck- | Checks for array index values in range |

**Table G.1** *(continued)*

| Metacommand | Action |
|---|---|
| $initck- | Checks for use of uninitialized values |
| $line- | Generates line number calls for debugger |
| $linesize:79 | Sets width of source listing |
| $list+ | Turns on or off source listing |
| $mathck- | Checks for mathematical errors |
| $message | Displays a message on terminal screen |
| $nilck- | Checks for bad pointer values |
| $ocode+ | Turns on or off object code listing |
| $page+ | Skips to next page |
| $page:*n* | Sets page number for next page |
| $pageif:*n* | Skips to next page if less than *n* lines left |
| $pagesize:55 | Sets page length of source listing |
| $pop | Restores saved value of all metacommands |
| $push | Saves current value of all metacommands |
| $rangeck- | Checks for subrange validity |
| $real | Sets the length of the REAL type |
| $rom | Warns on static initialization |
| $runtime- | Determines context of runtime errors |
| $simple | Disables global optimizations |
| $size | Minimizes size of code generated |
| $skip:*n* | Skips *n* lines or to end of page |
| $speed | Minimizes execution time of code |
| $stackck- | Checks for stack overflow at entry |
| $standard | Enables standard level only |
| $subtitle:'*subt*' | Sets page subtitle |
| $symtab+ | Sends symbol table to source listing |
| $system | Adds extend and system level features |
| $tagck- | Checks tag fields in variant records |
| $title:'*title*' | Gives page title for source listing |
| $warn+ | Gives warning messages in source listing |

# Index

**MICROSOFT®**
10700 Northup Way, Bellevue, WA 98004

# Software Problem Report

Name ______________________________

Street ______________________________

City ______________ State __________ Zip __________

Phone ______________________ Date __________

## Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

## Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem
(Document #__________)

_____ Other

## Software Description

**Microsoft Product** ______________________________

Rev. __________ Registration # ____________________

Operating System ______________________________

Rev. __________ Supplier ____________________

Other Software Used ______________________________

Rev. __________ Supplier ____________________

**Hardware Description**

Manufacturer ______________ CPU __________ Memory ______ KB

Disk Size ______″ Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals ______________________________

## Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

**Microsoft Use Only**

| | |
|---|---|
| Tech Support ____________ | Date Received ____________ |
| Routing Code ____________ | Date Resolved ____________ |
| Report Number ____________ | |

Action Taken:

MICROSOFT®