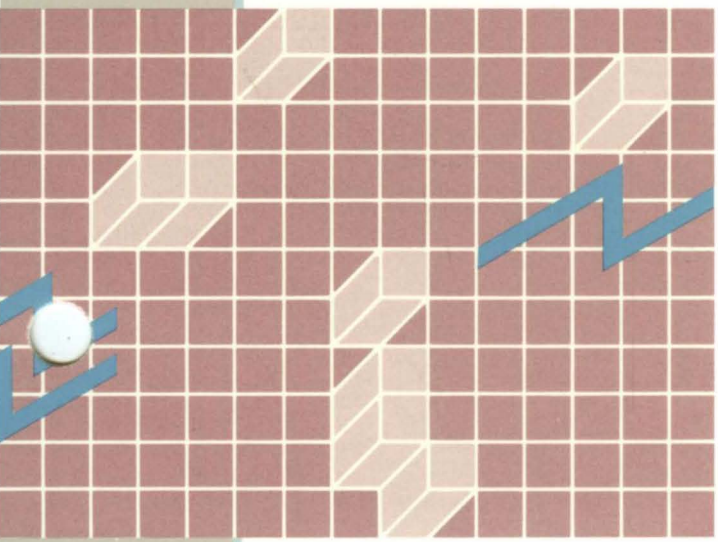
Pascal Reference

Volume 1



PASCAL REFERENCE MANUAL: VOLUME 1

Specifications Subject to Change.

Convergent Technologies, Convergent, CTOS, CT-BUS, CT-DBMS, CT-MAIL, CT-Net, DISTRIX, AWS, IWS, and NGEN are trademarks of Convergent Technologies, Inc.

CP/M-86 is a trademark of Digital Research.
MS, GW and XENIX are trademarks of Microsoft Corp.
UNIX is a trademark of Bell Laboratories.

Third Edition (September 1984) A-09-00852-01-A

Copyright © 1981, 1984
by Convergent Technologies, Inc.

All rights reserved. Title to and ownership of the documentation contained herein shall at all times remain in Convergent Technologies, Inc. and/or its suppliers. The full copyright notice may not be modified except with the express written consent of Convergent Technologies, Inc.

CONTENTS: VOLUME 1

LIST OF TABLES

SUMMARY OF CHANGES

The 9.Ø Release of Pascal is described in this third edition of the Pascal Manual (Volume 1, A-Ø9-ØØ852-Ø1-A, and Volume 2, A-Ø9-ØØ868-Ø1A).

This edition has been completely rewritten. Changes encompass descriptions of features added to the 7.Ø to 9.Ø releases, as well as the addition of more reference material. The User Manual and Report by Jensen and Wirth is no longer distributed with the manual.

Although the manual is still a reference manual and does not attempt to teach Pascal, much more information has been included in the manual to describe the use of standard Pascal. Care has been taken to point out how our version differs from the ISO standard. In addition, Appendix B summarizes how this version of Pascal differs from the ISO standard and from other popular versions of Pascal.

Features of our version of Pascal that probably will not be portable to versions supplied by other vendors are described in this manual as **extend level** features. **Extend level** is shown in boldfaced type to make it easy for you to find.

In addition more information has been added on the Pascal run-time routines, the compiler, and on the use of Pascal with our operating system. Sections 18 and 19, "Using the Compiler" and "Run Time and Debugging," respectively, and Appendix F, "Using Pascal as a Systems Programming Language," contain most of this information.

Features that have been added to Pascal since the second edition of the manual was published are summarized below.

- New numeric data types: INTEGER1, INTEGER2, INTEGER4, REAL4, AND REAL8 have been added. These are discussed in Section 5, "Simple Types."

- Real numbers are represented in 8087/IEEE format. Internal representations of all data types are discussed in Appendix G, "Internal Representations of Data Types," and conversion of old real number formats to the IEEE format is discussed in Appendix E, "Conversion to and from IEEE format."

- An 8087 instruction emulator has been added. The 8087 emulator will not be linked with your Pascal programs if the programs do not use any floating point constants or variables. However, it is automatically linked if they are used. The 8087 emulator is discussed in Section 18, "Using the Compiler."

- Operations on sets with up to 16 elements now generate inline code. Operations on sets also now use the stack instead of the heap for temporaries. Sets are discussed in Section 6, "Arrays, Records, and Sets."

- New predefined types have been added: ADSMEM, and ADRMEM. These are discussed in the subsection "Address Types" in Section 8, "Reference and Other Types."

- A new parameter type, CONSTS, is available. Use of CONSTS parameters parallels use of VARS for formal reference parameters. CONSTS parameters are discussed in the subsection, "Reference Parameters," in Section 8, "Reference and Other Types," and in the subsection "Constant and Segment Parameters" in Section 13, "Introduction to Procedures and Functions."

- The VALUE section now allows initialization of ADR and ADS variables. The VALUE section is discussed in Section 10, "Variables and Values."

- The operator XOR has been implemented and is discussed in Section 11, "Expressions."

- You can now turn off the default PUBLIC attribute of procedures and functions in a MODULE by using empty brackets ([]) in the MODULE header. This is discussed in Section 13, "Introduction to Procedures and Functions."

- Intrinsics are discussed in Section 14, "Available Procedures and Functions."

  - The following are new intrinsics: BYWORD, FILLSC, MOVRSL, MOVRSR, LOWORD, HIWORD, BYLONG.

  - The following intrinsics have been redefined (renamed and changed):

    COPY to COPYLST
    COPYSTR (new arguments only)
    FILLCHAR to FILLC
    MOVELEFT to MOVEL
    MOVERIGHT to MOVER
    POS to POSITN

  - Run-time intrinsics that used to take VAR parameters now also accept VARS parameters with these (permanent) exceptions: files, the LSTRING parameter to Encode and Decode, all parameters to ReadSet, and the msg parameter to Abort.

- Two new heap functions have been added: PREALLOCHEAP and PREALLOCLONGHEAP. Using the heap is discussed in the subsection "Dynamic Allocation in Section 14, "Available Procedures and Functions," and details are given for each function further on in that section.

  PREALLOCHEAP lets you specify how much storage is to be allocated for the heap, so that the remainder of short-lived memory can be used for other purposes.

  PREALLOCLONGHEAP performs the same function for the long heap.

- The long heap is available to user programs together with functions to handle its allocation: ALLMQQ, FREMQQ, GETMQQ, DISMQQ.

- Two new metacommands have been added: $REAL:N and $INTEGER:N. Metacommands are discussed in Section 17, "Metacommands."

- Compiler capabilities have been increased.
  - Compiler speed is improved; the compiler is about 30 percent faster.
  - Code generation is improved; Code density has improved 3 to 5 percent.
  - There has been a 100 percent increase in symbol table capacity. The Pascal compiler now compiles larger programs.
  - The compiler now uses the long heap (a heap that can be greater than one segment) for identifier storage, so larger programs can be compiled.
- Compiler and run-time error codes have been updated and are listed in Appendix A, "Compiler Error Messages."

RELATED DOCUMENTATION

The following manuals, or related products, are referenced in this manual. It may be helpful to have copies of them on hand when you are using this manual.

The complete Guide to Technical Documentation is provided in the Executive Manual or similar command-line interpreter manual for your operating system.

Assembly Language Manual

CTOS™ Operating System Manual

Debugger Manual

Executive Manual

Linker/Librarian Manual

Status Codes Manual

The Assembly Language Manual specifies the machine architecture, instruction set, and programming at the symbolic instruction level.

The CTOS™ Operating System Manual describes the operating system. It specifies services for managing processes, messages, memory, exchanges, tasks, video, disk, keyboard, printer, timer, communications, and files. In particular, it specifies the standard file access methods: SAM, the sequential access method; RSAM, the record sequential access method; and DAM, the direct access method.

The Debugger Manual describes the Debugger, which is designed for use at the symbolic instruction level. It can be used in debugging FORTRAN, Pascal, and assembly-language programs. (COBOL and BASIC, in contrast, are more conveniently debugged using special facilities described in their respective manuals.)

The Executive Manual describes the command interpreter, the program that first interacts with the user when the system is turned on. It describes available commands and discusses command execution, file management, program invocation, and system management. It also addresses status inquiry, volume management, the printer spooler, and execution of batch jobs. This manual now incorporates the System Utilities and Batch Manuals.

The Linker/Librarian Manual describes the Linker, which links together separately compiled object files, and the Librarian, which builds and manages libraries of object modules.

The Status Codes Manual contains complete listings of all status codes, bootstrap ROM error codes, and CTOS initialization codes. The codes are listed numerically along with any message and an explanation.

1 LANGUAGE OVERVIEW

This section presents an overview and summary of the elements of the Pascal language and their function, as implemented for our version of Pascal. It briefly discusses the compiler and available metacommands, programs and compilable parts of programs, procedures and functions, statements, expressions, variables, constants, types, identifiers, and notation. The remaining sections of the manual discuss each of these elements in more detail.

Note that this manual does not attempt to teach Pascal, but is intended as a reference for those who already have some familiarity with the language.

COMPILER

The Pascal system consists of the Pascal compiler and a library containing the Pascal run-time environment. A Pascal program is run by compiling its one or more source modules, linking the resulting object files with the Pascal library using the Linker, and invoking the resulting run file, which is usually done through the Executive.

The Pascal compiler translates your Pascal source programs into object modules. The compiler provides a source listing, error messages, and a number of compiler metacommands to aid in program development and debugging.

The compiler generates native 8086 machine code, which is directly executed by the hardware.

LANGUAGE LEVELS

This version of Pascal offers two language levels:

- The standard level is limited to features that conform to the ISO standard. Programs you create at this level are portable to and from other machines running other ISO compatible Pascal compilers.

- The extend level includes features specific to our version of Pascal. Programs that use extend level features may not be portable.

  Whenever extend level features are discussed in this manual the words **extend level** appear in boldface type to make them easy for you to locate.

THE COMPILER METACOMMANDS

The Pascal metacommands provide a control language for the compiler. They 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 run-time error checking code.

Metacommands are inserted inside comment statements. All the metacommands begin with a dollar sign ($).

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

A complete list of the available metacommands and a detailed description of how to use each can be found in Section 17, "Metacommands."

PROGRAMS AND COMPILABLE PARTS OF PROGRAMS

The compiler processes programs, modules, and implementations of units. Modules and implementations of units contain subroutines that you can compile separately and use in Pascal. Collectively, these compilable programs and parts of programs are referred to as compilands. You can compile modules and implementations of units separately and later link them to a program without having to recompile the module or unit.

(See Chapter 16, "Compilable Parts of a Program," for a complete discussion of programs, modules, and units. In addition, see Appendix H, "Programming Examples," for examples of complete Pascal programs.)

PROGRAM STRUCTURE

The fundamental unit of compilation is a program. A program has three parts, which occur in the following order:

1.	Program heading	Identifies the program and gives a list of program parameters.
2.	Declaration section	Contains declarations of labels, constants, types, variables, functions, and procedures. These must all be declared here in the declaration section (unless they are predeclared) before they are used in the body of the program.
3.	Body	Contains all the executable instructions that are not part of a procedure. 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.

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

The following program illustrates the three-part program 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.
```

MODULES

Modules are program-like units of compilation that contain the declaration of variables, constants, types, procedures, and functions. 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.

UNITS

A unit has two sections: an interface and an implementation. Like a module, a unit can 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.

- The implementation contains the actual instructions for the procedures and functions defined by the unit.

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;
VAR I : INTEGER;
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.

ADVANTAGES OF BREAKING PROGRAMS INTO MODULES AND UNITS

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

- If a program is large, breaking it into parts makes it easier to develop, test, and maintain.
- If recompiling the entire source file for a large program is time consuming, breaking the program into parts saves compilation time.
- If you intend to include certain routines in a number of different programs, you can create a single module or unit that contains these routines and then link it to each of the programs in which the routines are used.
- If certain routines are executed very frequently, you might place them in a module or unit to test the validity of an algorithm and later create and implement similar routines in assembly language to increase the speed of the algorithm.

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 subprograms invoked as statements.

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 is invoked as a part of an expression and 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 Section 13, "Introduction to Procedures and Functions," for a complete discussion of procedures and functions.

See Section 14, "Available Procedures and Functions," and Section 15, "File-Oriented Procedures and Functions," for a discussion of procedures and functions that are predeclared as part of this version of the Pascal language.

STATEMENTS

Statements perform actions, such as computing, assigning, altering the flow of control, and reading and writing files. Statements are used in the bodies of programs, procedures, and functions and are executed as a program runs. Statements available with this version of Pascal perform the actions shown in Table 1-1.

See Section 12, "Statements," for a detailed discussion of each of these statements.

Table 1-1. Summary of Statements. (Page 1 of 2)

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.

Table 1-1. Summary of Statements. (Page 2 of 2)

Statement	Purpose
Procedure	Invokes a procedure with actual parameter 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.

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 (the values 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:

- Arithmetic expressions perform arithmetic operations on the operands in the expression.
- Boolean expressions perform logical and comparison operations with Boolean results.
- 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, the following expression evaluates to a REAL result:

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

Expressions may also include function designators as shown in the example below:

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

ADDREAL is a function that has been previously declared to return a REAL value 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 Section 11, "Expressions," for a detailed discussion of expressions.

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. (See the subsection, "Types," for a discussion of data types.)

You declare a variable in the heading or declaration section of a compiland, procedure, or function. Once you have declared a variable, you can

- initialize it, in the VALUE section of a program
- assign it a value, with an assignment statement
- pass it as a parameter to a procedure or function
- use it in an expression

See Section 10, "Variables and Values," for a complete discussion of variables.

The VALUE section, a feature of this version of Pascal, applies only to statically allocated variables (variables with a fixed address in memory.)

To use the VALUE section, you first 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. For example:

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

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

- a number, such as 1.234 and 100
- a string enclosed in single quotation marks, such as 'Miracle' or 'A1207'
- 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 this version of Pascal, you can declare constants anywhere in the declaration section of a compiland, 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 of our version of Pascal are structured constants and constant expressions.

- VECTOR, in the following example, is an array constant:

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

- 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 Section 9, "Constants," for a complete discussion of these and other aspects of constants.

TYPES

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

- A simple data type represents a single value and includes
  - INTEGER
  - WORD
  - CHAR
  - BOOLEAN
  - enumerated
  - subrange
  - REAL (REAL4 and REAL8)
  - INTEGER4
- A structured data type represents a collection of values and includes
  - ARRAY
  - RECORD
  - SET
  - FILE
- A reference type allows recursive definition of types.

All variables in Pascal must be assigned a data type. A type is either predeclared (for example, INTEGER and REAL) or defined in the declaration section of a program. The following sample type declarations create types that can store information about a student:

```
TYPE
  SEXTYPE = (MALE, FEMALE); {enumerated type}
  PEOPLETYPE = RECORD       {record type}
      SEX     : SEXTYPE;
      AGE     : INTEGER
      END;
  POPULATIONTYPE = ARRAY [1..100] OF
                   PEOPLETYPE ; {array type}
```

For a detailed discussion of data types, see the following sections: Section 4, "Introduction to Data Types;" Section 5, "Simple Types;" Section 6, "Arrays, Records, and Sets;" Section 7, "Files;" and Section 8, "Reference and Other Types."

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, data types, and variables usually are given identifiers, but are not required to have them.

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 (Ø-9), or underscore characters(_). The compiler ignores the case of letters; thus, "A" and "a" are equivalent.

The underscore character is significant in our version of Pascal. Thus, the following are not identical:

```
FOREST

FOR_EST
```

The compiler considers only the first 31 characters of an identifier to be significant.

An identifier cannot be the same as a Pascal reserved word. (See the subsection "Reserved Words" in Section 2, "Notation," for a discussion of reserved words, and Appendix E, "Summary of Reserved Words and Predeclared Identifiers," for a complete list.)

See Section 3, "Identifiers," for a complete discussion of identifiers.

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. Characters are used as components of identifiers, separators, punctuation, or operators. Characters not part of the notation can still be used in a string literal or comment.

A good understanding of this notation will increase your productivity by reducing the number of subtle syntactic errors in a program. See Section 2, "Notation," for a detailed discussion of notation.

Also see Appendix C, "Pascal Syntax Diagrams," for specific information about Pascal syntax.

2 NOTATION

All components of this version of the Pascal language are constructed from the standard ASCII character set. Characters make up lines, each of which is separated by a carriage return (ØAh). Lines make up files.

You can use upper and lower case characters; however, the difference in case is not significant for identifier names.

Any individual character or any group of characters falls into one or more of the following four categories:

- identifiers or components of identifiers
- separators
- special symbols
- unused characters

Each of these categories is discussed below.

COMPONENTS OF IDENTIFIERS

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

This section discusses only how to construct identifiers. Their use is described thoroughly in Section 3, "Identifiers."

Identifiers must begin with a letter; subsequent components may include letters, digits, and underscore characters. Only the first 31 characters of an identifier are significant. If an identifier has more than 31 characters, the compiler gives a warning that the identifier is too long and has been truncated.

LETTERS

You can use uppercase and lowercase letters in identifiers in your source program. However, internally, the compiler converts all lowercase letters used in identifiers to the corresponding uppercase letters internally.

DIGITS

Digits in Pascal are the numbers zero through nine (Ø-9). Digits can occur in identifiers (for example, AS129M) or in numeric constants (for example, 1.23 and 456).

USING THE UNDERSCORE CHARACTER

The underscore (_) is the only nonalphanumeric character allowed in identifiers, and for this compiler the underscore is a significant character.

You can use the underscore to improve readability of identifier names in the same way you would use a space. For example, the identifiers in the right-hand column below are easier to read than those in the left-hand column.

POWEROFTEN	POWER_OF_TEN
MYDOGMAUDE	MY_DOG_MAUDE

You can also make identifiers more readable by using capitals for significant letters:

PowerOfTen	MyDogMaude

SEPARATORS

Separators delimit adjacent numbers, reserved words, and identifiers, none of which can have a separator embedded within it.

A separator can be any of the following ASCII characters:

- a space (ASCII code 20h) ( )
- a tab (ASCII code 09h) (→)
- a formfeed (ASCII code 0Ch) (↖)
- a linefeed (ASCII Code 0Ah) (↙)
- a comment

Always use a separator between an identifier and a number. If you fail to do so, the compiler generally issues an error or warning message. In a few cases, however, a missing separator can be accepted.

For example, at the **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.

COMMENTS

Any character can be used within a comment or string literal.

Comments in standard Pascal can span more than one line and take one of the following forms:

```
{This is a comment, enclosed in braces}

(*This is another form of comment*)
```

At the **extend level**, you can also begin comments with an exclamation point (!). For comments in this form, the linefeed delimits the comment.

Nested comments are permitted, 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. Note that, such nested comments might not be portable.

SPECIAL SYMBOLS

Special symbols fall into three categories:

- o punctuation
- o operators
- o reserved words

PUNCTUATION

Table 2-1 below summarizes the use of several punctuation symbols.

Table 2-1. Summary of Punctuation. (Page 1 of 2)

Symbol	Purpose
{ }	Braces delimit comments.
[ ]	Brackets delimit array indices, sets, and attributes. They can also replace the reserved words BEGIN and END in a program.
( )	Parentheses delimit expressions, parameter lists, and program parameters.
'	Single quotation marks delimit string literals.
:=	A colon directly followed by an equal sign denotes an assignment statement.
;	A semicolon separates statements and declarations.

Table 2-1. Summary of Punctuation. (Page 2 of 2)

:	A colon separates variables from types and labels from statements.
=	An equal sign separates identifiers and type clauses in a TYPE section.
,	A comma separates the components of a list.
..	A double period denotes a subrange.
.	A period designates the end of a program, indicates the fractional part of a real number, and delimits fields in a record.
^	A caret denotes the value pointed to by a reference value. The question mark (?) and the at sign (@) are synonyms for the caret.
?	The question mark denotes the value pointed to by a reference value.
@	The at sign denotes the value pointed to by a reference value.
#	A number sign denotes nondecimal numbers.
$	A dollar sign prefixes metacommands.

OPERATORS

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

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

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

Some operators are reserved words, for example, NOT and DIV. (See below).

See Section 11, "Expressions," for a complete list of the nonalphabetic operators and a discussion of the use of operators in expressions.

RESERVED WORDS

Reserved words are used for names of attributes, directives, and features of the standard and extend levels of Pascal.

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

Reserved words are a fixed part of our version of Pascal. They include, for example, statement names, such as BREAK, and words such as BEGIN and END that bracket the main body of a program.

See Appendix D, "Summary of Reserved Words and Predeclared Identifiers," for a complete list of reserved words.

UNUSED CHARACTERS

The following printing characters are not used by this version of Pascal:

```
%   &   |
```

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

Error messages are generated if you use the following nonprinting ASCII characters in anything but a comment or string literal in a source file:

- ASCII characters Ø to 31 (Øh to 1Fh), excepting the tab character (Ø9h) and the formfeed character (ØCh)
- ASCII characters 127 to 255 (7Fh to ØFFh)

The tab character is treated as a space and is passed to the listing file. A formfeed is treated as a space and starts a new page in the listing file.

OTHER NOTES ON CHARACTERS

As an extension to the ISO standard, the question mark (?) or the at sign (@) can be substituted for a caret (^).

Table 2-2 gives a list of pairs of printing characters that represent the same ASCII character.

Table 2-2. Equivalent ASCII Characters.

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

3 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, data types, and variables usually are given identifiers, however they are not required to have them.

Some identifiers are predeclared; others you declare in a declaration section. Standard Pascal allows identifiers for the following elements of the Pascal language:

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

The following **extend level** features also can have 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 are significant, for example MY_IDENTIFIER.

Identifiers can be as long as you wish, as long as they fit on a single line. However, only the first 31 characters of an identifier are significant. The compiler generates a warning message, not a error, when an identifier longer than 31 characters is encountered.

Standard Pascal allows unsigned integers as statement labels.

Extend level Pascal allows labels that are normal alphabetic identifiers.

Statement labels have the same scope rules as identifiers. (See the subsection "The Scope of An Identifier," below.) Leading zeros are not significant.

Identifiers of seven characters or fewer save space during compilation.

NOTE

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

In addition, PUBLIC names should not begin with the characters XXX, as this can cause problems when the program is linked.

DECLARING AN IDENTIFIER

You declare identifiers in the declaration section of a program, module, interface, implementation, procedure, or function. You can also declare identifiers in the heading of a program, procedure, or function. The declaration associates the identifier with a language object. Table 3-1, shows examples of identifiers that might be used for each of the possible language objects, and gives examples of the syntax used for the declaration.

Table 3-1. Declaring Identifiers.

Object	Identifier	Sample Declaration
Program	Z	PROGRAM Z (INPUT,OUTPUT)
Module	ABC	MODULE ABC
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	A	LABEL A
Label	HAWAII	LABEL HAWAII
Procedure	BANG	PROCEDURE BANG
Function	FOO	FUNCTION FOO: INTEGER

THE SCOPE OF AN IDENTIFIER

An identifier is defined for the duration of the program, module, implementation, interface, procedure, or function in which you declare it. This holds true for any nested procedure or function. 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 the subsection "Notes on Reference Types" in Section 8, "Reference and Other Types."

PREDECLARED IDENTIFIERS

Our version of Pascal makes available a number of predeclared identifiers, which you can use freely without declaring. Predeclared identifiers differ from reserved words in that you can redefine them whenever you wish.

Predeclared identifiers include the identifiers for predeclared types, super array types, constants, file variables, functions, and procedures. See Appendix D, "Summary of Reserved Words and Predeclared Identifiers," for a list of the predeclared identifiers for this version of Pascal.

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 1..1Ø is declared explicitly. An explicitly declared type can also be given a type identifier. In the latter case, a type declaration is required.

Types fall into three broad categories: simple, structured, and reference.

- Simple types, for example INTEGER, cannot be divided into any other types.
- Structured types, however, are composed of other types that can be structured or simple, themselves. An example of a structured type is an array, which can be a collection of integers, as in ARRAY [1..1Ø] OF INTEGER.
- Reference types allow data structures that vary in size and form and provide an indirect form of access. An example of a reference type is the predeclared address type ADR, which means an actual machine address, a 16-bit offset into the default data segment.

Table 4-1 below gives a breakdown of the types in each of these categories. The remainder of this section provides an introduction and discusses types in general; each category is discussed in detail in Sections 5 through 8.

Table 4-1. Categories of Types.

Category	Types Included	Comments/Examples
Simple	Ordinal	
	INTEGER	-MAXINT..MAXINT
	WORD	Ø..MAXWORD
	CHAR	CHR(Ø)..CHR(255)
	BOOLEAN	(FALSE,TRUE)
	enumerated	e.g., (RED,BLUE)
	subrange	e.g., 1ØØ..5ØØØ
	REAL4, REAL8	
	INTEGER4	-MAXINT4..MAXINT4
Structured	ARRAY OF type	
	general (OF any type)	
	STRING(n)	[1..n] of CHAR
	LSTRING(n)	[Ø..n] of CHAR
	RECORD	
	SET OF type	
	FILE OF	
	general (binary) files	
	TEXT	Like FILE OF CHAR
Reference	Pointer	for example, ^TREETIP
	ADR OF type	Relative address
	ADS OF type	Segmented address
Procedural and Functional		Only as parameter type
Super Array	SUPER ARRAY OF type	
	general (OF any type)	
	STRING	[1..*] of CHAR
	LSTRING	[Ø..*] of CHAR

DECLARING DATA TYPES

The type declaration associates an identifier with a type of value. You declare types in the TYPE section of a program, module, interface, implementation, procedure, or function. Types are not declared 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 COLOR = (RED, BLUE, GREEN);
     NAMES = (TOM, DICK, HARRY);
       AGE = 0..60;
```

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 PLAYER : NAMES;
    BALL : COLOR;
```

The examples below are also type definitions. PAGE is a structured type that contains other structured types.

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

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 are a standard exception to this rule and are discussed in Section 8, "Reference and Other Types."

Super types are a special feature of our version of Pascal. A super type is like a set of types or like a function that returns a type.

The only super types currently available are super arrays. (Super arrays are discussed in the subsection of that name in Section 6, "Arrays, Records, and Sets."

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.

TYPE COMPATIBILITY

To the ISO standard for type compatibility, our version of Pascal adds rules for super array types, LSTRINGs, and constant coercions (that is, forced changes in the type of a constant). Type transfer functions, to override the typing rules, are also available.

Two types can be "identical," "compatible," or "incompatible." An expression can be "assignment compatible" with a variable, value parameter, or array index.

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 such as the following:

```
TYPE T1 = T2;
```

"Identical" types are truly identical in this version of Pascal: there is no difference between types T1 and T2 in the example above. Type identity is based on the name of the types, rather than 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. (For an explanation of actual and formal reference parameters, see the Glossary.)

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 changes to any LSTRING type with a large enough bound. For example, the type of 'ABC' changes to LSTRING (5) if necessary.

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 assignment, comparisons, READs, and WRITEs, LSTRING (n) is not a shorthand notation for PACKED ARRAY [Ø..n] OF CHAR. The two types are not identical, compatible, or assignment compatible. (See the subsection "Using STRINGs and LSTRINGs," in Section 6, "Arrays, Records, and Sets," for further information on string types.)

TYPE COMPATIBILITY AND EXPRESSIONS

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

- They are identical.
- They are both ADR types.
- They are both ADS types.
- One is a subrange of the other.
- They are subranges of compatible types.

Two structured types are compatible if one of the following is true; if they are

- identical
- SET types with compatible base types
- STRING-derived types of equal length
- LSTRING-derived types

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

- Either type is a FILE or contains a FILE.
- Either type is a super array type.
- 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 Section 11, "Expressions," for details.) A CASE index expression type must be compatible with all CASE constant values. Note that two sets are never compatible if one is PACKED and the other is not.

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 (that is, DESTINATION := SOURCE is permitted) if one of the following is true:

- T_SOURCE and T_DEST are identical types.
- T_SOURCE and T_DEST are compatible and SOURCE has a value in the range of subrange type T_DEST.
- T_DEST is of type REAL or REAL8 and T_SOURCE is compatible with type INTEGER or INTEGER4.
- 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:

- for SETs, every member of SOURCE is in the base type of T_DEST
- for LSTRINGs,

```
UPPER (DESTINATION) >= SOURCE.LEN
```

(The predeclared function UPPER is discussed in Section 14, "Available Procedures and Functions.")

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

- passing value parameters
- READ and READLN procedures
- control variable and limits in a FOR statement
- super array type array bounds, and array indexes

If the range checking switch ($RANGECK) is on, assignment compatibility is checked at run time; otherwise, no checking is done. Assignment compatibility is usually known and checked at compile time. However, some subrange, set, and LSTRING assignments depend on the value of the expression to be assigned and thus cannot be checked until run time.

5 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 Section 6, "Arrays, Records, and Sets," and Section 7, "Files") are composed of other types. The simple data types fall into three categories:

- ordinal
- REAL
- INTEGER4

ORDINAL TYPES

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

- INTEGER
- WORD
- CHAR
- BOOLEAN
- enumerated
- subrange

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

INTEGER

INTEGER values are a subset of the whole numbers and range from -MAXINT through Ø to MAXINT. MAXINT is the predeclared constant 32767 (2**15 - 1). (The value -32768 is not a valid INTEGER; the compiler uses it to check for uninitialized INTEGER and INTEGER subrange variables.)

Note that INTEGER is not a subrange of INTEGER4 (discussed in the subsection "INTEGER4," below.) 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 can 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 REAL8. The ORD function converts a value of any ordinal type to an INTEGER type.

The predeclared type INTEGER2 is identical to INTEGER.

The predeclared type INTEGER1, has the same range as the data type SINT, -127 to +127. It occupies only one byte of storage, in contrast to INTEGER, which occupies two bytes.

WORD

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 Ø to MAXWORD (65535, 2**16 - 1).

The WORD type, a feature of our version of Pascal, is useful in several ways:

- to express values in the range from 32768 to 65535
- to operate on machine addresses
- 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:

- as a signed value ranging from -32767 to +32767
- as a non-negative value ranging from Ø to 65535

However, do not mix WORD and INTEGER values in an expression (although doing so generates a warning rather than an error message). Neither are WORD and INTEGER values assignment-compatible.

CHAR

In this version of 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.

The CHR function changes any ordinal type value to CHAR type, as long as ORD of the value is in the range from Ø to 255.

(See Appendix B, "Standard Character Set," in the CTOS Operating System Manual for a complete listing of the ASCII character set.)

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 Ø and ORD (TRUE) is 1. This means that FALSE < TRUE.

You can redefine the identifiers BOOLEAN, FALSE, and TRUE, but the compiler implicitly uses the former 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) <> Ø
```

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)
```

The type values (for example, RED, CLUB, or MAUDE) do not have to be declared in the CONST section or any other section in the program.

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

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 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)   = Ø
ORD (WHITE) = 1
ORD (BLUE)  = 2
```

The RETYPE function, at **extend 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')
```

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 any 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 can 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)
```

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 bound of the subrange (in that order). The lower bound must not be greater than the upper bound, but the bounds can be equal. The subrange type is frequently used as the index type of an array or as the base type of a set. (See Section 6, "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 can substitute a subrange clause for a list of values in the following circumstances:

- set constants
- set constructors
- 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 to guarantee that the value of a variable is within acceptable bounds. If the range-checking switch ($RANGECK) is on during compilation, these bounds are checked at run time.

For instance, if the logic of a program implies that a variable always has a value from 1ØØ 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 can permit the compiler to allocate less room and use simpler operations. For example, declaring BOTTLES to be the INTEGER subrange 1..1ØØ means that the type can be allocated in eight bits instead of sixteen.

Three subrange types are predeclared:

- BYTE = WRD(Ø)..255;
  {8-bit WORD subrange}

- SINT = -127..127;
  {8-bit INTEGER subrange}

- INTEGER1 = SINT

The BYTE type is particularly useful in machine-oriented applications. For example, the ADRMEM and ADSMEM types 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. (See the subsection, "Address Types," in Section 8, "Reference and Other Types" for details on ADRMEM and ADSMEM types.)

In some cases (for example, an assignment of a BYTE expression to a BYTE variable when the math-checking switch ($MATHCK) 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;
```

REAL

For real numbers, standard Pascal provides a type REAL. With this version of Pascal, three real types are available:

- o REAL4 — Single precision real numbers (7 significant digits)
- o REAL8 — Double precision real numbers (15 significant digits)
- o REAL — Identical to either REAL4 or REAL8

Note that the type REAL is always either REAL4 or REAL8. The choice is made with a metacommand, $REAL:n, where n is either 4 or 8. Thus, {$REAL:8} has the same effect as TYPE REAL = REAL8. The default type for REAL is REAL4.

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

This version of Pascal uses the IEEE real number format. 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:

- o REAL4 — Sign bit, 8-bit binary exponent with bias of 127, 23-bit mantissa
- o 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 7 significant digits (24 bits), with an exponent range of E-38 to E+38. The REAL8 numeric range includes over 15 significant digits (53 bits), with an exponent range of E-306 to E+306.

As an extension to standard Pascal, the exponent character can be "D" or "d" as well as "E" or "e", for example, 12.34d56. Note that the D or d exponent character does not indicate double precision, as it does in the FORTRAN language.

This version of Pascal performs floating point operations using either the 8087 math coprocessor chip, or the Pascal run-time support library 8087 emulator routines. The 8087 chip is an option installed only on some workstations. (See the subsection "Linking Your Program" in Section 18, "Using the Compiler," for information on how to link you program if you have an 8087 chip.)

Operations on two REAL4 operands are calculated in REAL4 precision with the 8087 emulator, but with REAL8 precision if you have an 8087 chip installed on your workstation.

REAL literals 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.

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 the subsection "Boolean Expressions," in Section 11, "Expressions," for a description of REAL comparisons that produce an unordered result.)

All results are rounded up to the nearest representable number (with 0.5 rounded up or down to make the next digit even.)

INTEGER4

As with INTEGER and WORD values, INTEGER4 values are a subset of the whole numbers. INTEGER4 values occupy four bytes of storage and range from -MAXINT4 to MAXINT4. MAXINT4 is a predeclared constant with the value of 2,147,483,647 (2**31 - 1). The value -2,147,487,648 (-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. INTEGER4 values also cannot be used to control FOR and CASE statements.

INTEGER4 is currently an extended numeric type, as is 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 FLOAT4 function to make the conversion. The functions ROUND4 and TRUNC4 are also available for REAL/INTEGER4 conversion.

To assign a WORD to an INTEGER4, use BYLONG instead of the ORD function, because ORD will sign-extend the sign bit of the WORD. For example:

```
Integer4Var := BYLONG (Ø, WordExpression);
```

6 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. A structured type can occupy up to 65534 bytes of memory.

The following are structured types:

- ARRAY <range> OF <type>
- SUPER ARRAY <range> OF <type>
  - STRING
  - LSTRING
- RECORD
- SET OF <base-type>
- FILE OF <type>
  - TEXT

Because components of structures can be structured types themselves, you can 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 section discusses arrays, records, and sets. See Section 7, "Files," for a discussion of files.

ARRAYS

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

The elements of the array are designated by indexes, 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:

```
VAR

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 shorthand notation available for n-dimensional arrays makes the following statement:

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

the same as

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

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

```
INT_ARRAY [10]  := 1234;

ARRAY_2D [0,8] := 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 Section 8, "Reference and Other Types," for a discussion of packed types.

SUPER ARRAYS

A super array is a special variable-length array. Using the super array type you can pass arrays of different lengths to a reference parameter and you can create dynamically dimensioned arrays.

A super array is an example of a 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 our version of Pascal.

A super type identifier specifies the set of types represented by the super type. A later type declaration can 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. The 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) denote simple derived types: VECTOR (10) denotes ARRAY [1..10] OF REAL and VECTOR (30) denotes ARRAY [1..30] OF REAL. 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, 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 Section 8, "Reference and Other Types," for a discussion of pointer types and dynamic allocation. See Section 14, "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 [Ø..*, Ø..*] OF
             REAL;
      .
      .
    NEW (STR_PNT, 12);
    NEW (VEC_PNT, 9, 99);
```

where 12, 9, and 99 replace the asterisks (*).

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}
```

```
VAR S: REAL; I: INTEGER;
BEGIN
  S := Ø;
  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 (that is, MAXINT, MAXWORD).

A super array type's internal representation is similar whether it is a reference parameter or the referent of a pointer. First comes the address (reference parameter) or pointer value, which is either 2 or 4 bytes long. Following the address are the upper bounds, which are signed or unsigned 16-bit quantities. The bounds occur in the same order as they are declared. A pointer value to a super array type is normally longer than other pointers, since the upper bounds are included.

Two super array types are predeclared: STRING and LSTRING. The compiler directly supports STRING and LSTRING types in the following ways:

- LSTRING and STRING assignment
- LSTRING and STRING comparison
- LSTRING and STRING READs
- access to the length of a STRING with the UPPER function

- access to maximum length of an LSTRING with the UPPER function
- access to LSTRING length with STR.LEN and STR[Ø], where STR is the identifier of the LSTRING

These subjects are discussed in the subsection "Using STRINGs and LSTRINGs."

The following are some of the most powerful ways you can use super arrays:

- To process strings.

  Both STRING and LSTRING are predeclared super array types. The LSTRING type handles variable-length strings, with a maximum length of 255 characters. STRING handles fixed-length strings, including strings more than 255 characters long.

- To dynamically allocate arrays of varying sizes.

  Otherwise, such arrays would need a maximum possible size preallocation.

- As a 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.

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), which you cannot do with other arrays.

In this version of Pascal, the super array notation STRING (n) is identical to PACKED ARRAY [1..n] OF CHAR. (n can 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 cannot 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 the subsection, "Using STRINGs and LSTRINGs."

Standard Pascal supports assigning, comparing, and writing 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 normal 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 can vary, comparisons and assignments of STRING variables 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 can 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 our version of Pascal, however, this restriction does not apply.

To conform with the ISO standard, passing of the CHAR component of a STRING as a reference parameter is defined as an "error not detected." 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 the subsection, "LSTRINGs."

A number of intrinsic procedures and functions for strings are discussed in Section 14, "Available Procedures and Functions." Many of the procedures and functions described work on STRINGs; some apply only to LSTRINGs.

LSTRINGs

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

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

However, although they are structurally the same, a variable with the explicit type PACKED ARRAY [0..n] OF CHAR is not "identical" to the type LSTRING (n). There is no default for n, the range of which 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, if required.

The predeclared constant, NULL, is the empty string, LSTRING (Ø). 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. An LSTRING can be assigned to any other LSTRING, if 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, if the current length of the actual parameter is not greater than the maximum length specified by the formal parameter. If the range-checking switch ($RANGECK) is on, the compiler generates code to check 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';
    {String literal on right converted to }
    {LSTRING(A)}
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 can compare any two LSTRINGs, including super array type LSTRINGs. (This comparison is 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.

USING STRINGs AND LSTRINGs

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

See also Section 14, "Available Procedures and Functions," for descriptions of the following string procedures and functions:

CONCAT	INSERT
COPYLST	POSITN
COPYSTR	SCANEQ
DELETE	SCANNE

At the **extend level**, the procedures FILLC, FILLSC, MOVEL, MOVESL, MOVER, and MOVESR also operate on strings.

This version of Pascal supports STRINGs and LSTRINGs directly in the following ways:

Assignment — You can 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, if the range-checking switch is on ($RANGECK), a run-time error occurs. You can 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 done similarly to making an assignment.

Comparison

The LSTRING operators < <= > >= <> = use the length byte for string comparisons; the operands can 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.

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 Section 15, "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 the string 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.

Length access

You can access the current length of an LSTRING variable T with T.LEN, which is of type BYTE, or with T[Ø], which is of type CHAR. This notation can be used to assign a new length, as well as determine the current length. The UPPER function finds 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, STRING and CHAR constants can be used as if they were LSTRING constants. NULL (the zero length LSTRING) is the only explicit LSTRING constant.

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

You can pass an actual LSTRING parameter to a formal reference parameter of type STRING using the following technique. 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.

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 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.}
LTR5VAR := CHR1VAR;
{Character variable is not OK.}
LTR5VAR := 'TINY';
{Smaller STRING constant is OK.}
LTR5VAR := 'RIGHT';
{Same length STRING constant is OK.}
LTR5VAR := 'longer';
{This gives an error at run time only; OK for}
{now.}
LST5VAR := LST9VAR;
{This can give an error at run time; OK for}
{now.}
LST9VAR := LST5VAR;
{This isn't even checked at run time; 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.
```

RECORDS

A record structure acts as a template for a collection of 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 either record notation or the WITH statement.

The example below declares a record with three fields: TITLE of type LSTRING(100), ARTIST of type LSTRING (100), and PLASTIC of type ARRAY [1..SONG_NUMBER] OF SONG_TITLE:

```
CONST SONG_NUMBER = 1000;

TYPE
      SONG_TITLE = LSTRING (20);

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

In this example, when a variable of type LP is declared (for example BEATLES_1, below), the compiler allocates a contiguous block of memory sufficient to hold all the fields. Each field then can be used as an independent variable.

```
VAR BEATLES_1 : LP;
```

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

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

Thus, BEATLES1.TITLE can be used as a variable of type LSTRING(100). Note that when the WITH statement is used above, using PLASTIC within it is identical to using BEATLES1.PLASTIC outside the context of the WITH statement.

VARIANT RECORDS

Variant records are records where some fields can share memory (that is, overlap). Consider the declarations

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

Suppose a variable of type OBJECT is to hold its SIZE and ANGLE (some parameters) if S is equal to SQUARE, and it is to hold DIAMETER if S is equal to CIRCLE. In the former case, DIAMETER is ignored; in the latter case SIZE and ANGLE are ignored.

You can make the pair SIZE and ANGLE share memory with DIAMETER as follows:

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

The declaration for S can be made part of the CASE, as below. The parentheses used are mandatory:

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

S is called a tag field. If a variable is declared of type OBJECT (for example, VAR OB : OBJECT), then the variable is given enough memory to hold either SIZE and ANGLE or DIAMETER, (but not all three), and the contents of the shared memory are interpreted according to the value of the tag field (S). The fields are accessed as usual.

Examples:

```
OB.X := 1.Ø;

IF OB.S = SQUARE
THEN OB.SIZE := 3.Ø
ELSE OB.DIAMETER := 4.Ø;

WITH OB DO
BEGIN
  S := CIRCLE;
  DIAMETER := 3.1
END;
```

Thus, when you use OB.SIZE, OB.S should equal square, but the compiler does not check that.

The tag field may have no memory allocated for it and no identifier either. That is, you can have

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

This declaration is similar to the previous ones: the variables for this type will hold the values of X,Y, and either SIZE and ANGLE or DIAMETER (but not all three). No memory is allocated in the record for the tag field, so consistent use of the variants (SIZE, ANGLE, DIAMETER) is entirely your responsibility.

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.

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 below, you could create and then assign to the variables shown:

```
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;

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 detected by the compiler.

Our version of 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:(). You can even declare an entirely empty record type, although the compiler issues a warning whenever the record is used.

```
TYPE R = RECORD
           I : INTEGER;
           CASE W : WORD OF
             1,2,5 : (RE : REAL);
               {1,2,5 is a list of constants}
             8..8Ø : (P,Q : WORD);
               {8..8Ø is a subrange of constant}
               {values}
             1ØØ : ()  {empty variant}
           END;
```

The ISO standard defines a number of errors that relate to variant records; these errors may not be detected by the compiler.

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, our version of 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 detected by the compiler.

Various restrictions are not enforced on a record variable allocated on the heap with the long form of the NEW procedure. (See Section 14, "Available Procedures and Functions," for details.) However, this version of Pascal does check an assignment to such a record to see that only the record itself is modified in the heap.

A record allocated with the long form of NEW can be released using the short form of DISPOSE with no ill effects. (This is an ISO error not detected by the compiler.) It is also an error not detected to DISPOSE of a record passed as a reference parameter or used by an active WITH statement.

Variant records interact with features of this version of Pascal in two ways:

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

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

- Giving initial data to several overlapping variants in a variable in a VALUE section can have unpredictable results. Note that the records field can be initialized in the VALUE section like any other variables. 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;
```

The compiler generates a warning message if you attempt either of these operations.

EXPLICIT FIELD OFFSETS

You can assign explicit byte offsets, from the beginning of a record, to the fields in a record. This **extend level** feature can be useful for interfacing to software in other languages, since their formats may not conform to Pascal's 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 [ØØ]: BYTE;
           {occupies byte Ø}
           FILENM [Ø1]: STRING (8);
           {occupies bytes 1 through 8}
           FILEXT [Ø9]: STRING (3);
           {occupies bytes 9 through 11}
           EXTENT [12]: BYTE;
           CPMRES [13]: STRING (2Ø);
           RECNUM [33]: WORD;
           RECOVF [35]: BYTE;
           END;

     OVERLAP = RECORD
               BYTEAR [ØØ]: ARRAY [Ø..7] OF
                            BYTE;
               {occupies bytes Ø through 7}
               WORDAR [ØØ]: ARRAY [Ø..3] OF
                            WORD;
               {occupies bytes Ø through 5}
               BITSAR [ØØ]: SET OF Ø..63;
               END;
```

As can be seen in the example, the offset is enclosed in brackets []; this is similar to attribute notation. The number is the byte offset of the start of the field.

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, since otherwise the fields may overlap unpredictably.

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));
```

(For more information on each of the procedures and functions described above, see the subsection "Dynamic Allocation" and the individual entry for each procedure or function in Section 14, "Available Procedures and Functions.")

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.

SETS

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

Suppose you declare the following set types:

```
TYPE COLOR = (RED, BLACK, WHITE, GREEN, BLUE);
     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 values to these set variables:

```
FLAG := [RED, WHITE, BLUE];
{a subset of COLOR}
VOWELS := ['A', 'E', 'I', 'O', 'U'];
{a subset of 'A..Z'}
LIVE := [ANIMAL, VEGETABLE];
{a subset of (ANIMAL, VEGETABLE, MINERAL)}
```

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 inline code or by routines in the set unit. See Section 11, "Expressions," for a complete discussion of operations on sets.

The ORD value of the base type can range from Ø to 255. Thus, SET OF CHAR is legal, but SET OF 1942..1984 is not. (See Section 14, "Available Procedures and Functions" for a discussion of ORD.)

Sets whose maximum ORD value is 15 (that is, sets that fit into a WORD) are usually more efficient than larger ones. If the range-checking switch ($RANGECK) is on, passing a set as a value parameter invokes a run-time 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) <> Ø) THEN WRITELN ('GO
FISH');
```

The equivalent declaration in this version of Pascal might be:

```
TYPE
QUALITIES = SET OF (READY, GETSET, ACTIVE,
              DONE);
VAR X : QUALITIES;
```

You could then assign the qualities with X := [READY, 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')
```

The number of bytes allocated for a SET is

(ORD (upperbound) DIV 16) * 2 + 2

You can also use SET OF Ø..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.

INTERNAL REPRESENTATION OF ARRAYS, RECORDS, AND SETS

For arrays and records, the internal form is comprised of the internal forms of the components, in the same order as in the declaration. Arrays, records, variants, sets, and files always start on a word boundary. In any case, variables cannot be allocated more than MAXWORD (64K-1) bytes.

A PACKED type has the same representation as an unpacked one.

A variable or component 16 bits or larger is always aligned on a word boundary; therefore, it always has an even byte address. The only exception is when explicit field offsets are given by the user in a program.

An 8-bit variable is also aligned on a word boundary, but an 8-bit component of a structure (array or record) is aligned on a byte boundary, which can be at an even or odd address. An array of 8-bit variables starts on a word boundary.

This is always an even number from 2 to 32 bytes. For example, SET OF 'A'..'Z' requires 12 bytes. Internally, a set consists of an array of bits, with one bit for every possible ORD value from Ø to the upper bound. Bits are assigned starting with the lowest addressed byte, and bits in each byte are assigned starting with the most significant bit. The occurrence of a given ORD value as an element of a set implies the corresponding bit is 1, and the byte and bit position of a given ORD value of any set is the same. For example, if 'A' is the base type of a set, the ORD value of 'A' is 65, and the second bit is of the ninth byte in a set is 1 if 'A' is in the set. For example, for [A], the ninth byte is 2#Ø1ØØØØØØ.

7 FILES

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

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 COLOR = (RED, BLUE);
     F1 = FILE OF COLOR;
     F2 = FILE OF CHAR;
     F3 = TEXT; {Similar to FILE OF CHAR}
```

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 usually corresponds to one of the following:

- disk files
- keyboard
- video display
- printers
- other input and output devices

This implies the following restriction in 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.

The operating system is used to access files, and no additional formatting or structure is imposed on the files.

Our version of Pascal supports normal statically allocated files, files as local variables (allocated on the stack), and files as pointer

referents (allocated on the heap). (File in this context, refers to a file control block, that is a memory structure that contains information about the file.) 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. (See Section 16, "Compilable Parts of a Program," for a discussion of units, modules, and interfaces.)

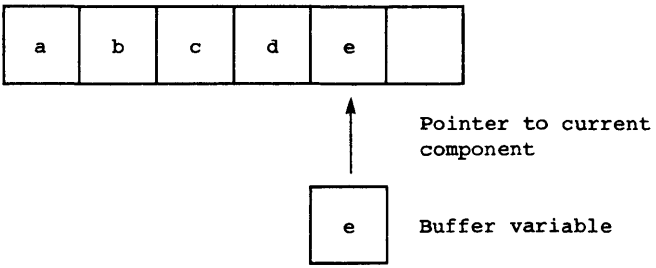
This version of Pascal sets up the standard files, INPUT and OUTPUT (discussed in the subsection, "INPUT and OUTPUT," below). In standard Pascal, files must be given in the program header; and when you run your program, the run-time system prompts you for filenames. At the **extend level**, you can use the ASSIGN or 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 can 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.

BUFFER VARIABLES

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 (that is, its value fetched or stored) like any other 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 might 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}
```

A special internal mechanism in this version of Pascal, lazy evaluation, allows interactive input in a natural way. Lazy evaluation is applied to all ASCII structured files and is necessary for natural input.

Lazy evaluation generates a run-time call that is executed before any use of the buffer variable. See the subsection, "Lazy Evaluation," in Section 15, "File-Oriented Procedures and Functions," for complete details.

FILE STRUCTURES

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

BINARY

The Pascal data type FILE OF <type> corresponds to BINARY structure files. These, in turn, correspond to unformatted operating system files. See the subsection "File Access Modes" for further discussion of BINARY files.

ASCII

The Pascal data type TEXT corresponds to ASCII structure files. These, in turn, correspond to textual operating system files, which we refer to as textfiles.

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.

Pascal textfiles (files of type TEXT) are divided into lines with a line marker, (the line feed: ØAh). When read, this character always looks like a blank.

At the **extend level**, a declaration for a textfile can include an optional line length. Setting the line length, which sets record length, is only needed for DIRECT mode textfiles. You can 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);
```

FILE ACCESS MODES

The file access modes are SEQUENTIAL, TERMINAL, and DIRECT. SEQUENTIAL and TERMINAL mode files are available at the standard level; at the **extend level**, DIRECT mode is also available. The default mode is SEQUENTIAL for all files except INPUT and OUTPUT, for which the default mode is TERMINAL.

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.

TERMINAL MODE FILES

TERMINAL mode files always correspond to an interactive terminal (keyboard and video display) or printer. TERMINAL mode files, like SEQUENTIAL mode files, are opened at the beginning of the file for either reading or writing. Bytes 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 video display while the line is being typed.

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 keyboard. See the subsection, "Lazy Evaluation," in Section 15, "File-Oriented Procedures and Functions," for details.

SEQUENTIAL MODE FILES

SEQUENTIAL mode files are generally disk files or other sequential devices that support sequential access. 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).

DIRECT MODE FILES

DIRECT mode files are generally disk files or other random access devices. DIRECT mode files and the ability to access the mode of a file are available at the **extend level.**

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.

INPUT AND OUTPUT

Two files, INPUT and OUTPUT, are predeclared in every program. These files get special treatment as program parameters (discussed in Section 16, "Compilable Parts of a Program") 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:

```
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 can redefine the identifiers INPUT and OUTPUT, the file assumed by textfile input and output procedures and functions (for example, 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 can also use these procedures explicitly).

INPUT and OUTPUT have ASCII structure and TERMINAL mode. They are initially connected to your keyboard and video display and opened automatically. At the **extend level**, you can change these characteristics if you wish.

EXTEND LEVEL I/O

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

Along with access to certain FCB fields, **extend level** I/O also includes the following procedures:

ASSIGN	READFN
CLOSE	READSET
DISCARD	SEEK

See the subsection, "Extend Level I/O," in Section 15, "File-Oriented Procedures and Functions" for a description of these procedures.

The following types are predeclared:

```
TYPE FILEMODES = (SEQUENTIAL, TERMINAL,
                  DIRECT);
     FCBFQQ = RECORD
              TRAP : BOOLEAN;
              ERRS : BYTE;
              MODE : FILEMODES
              END;
```

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 can change or examine these fields at any time.

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 the keyboard, video, or

printer. 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.

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 to FALSE. If FALSE and an I/O error occurs, the program aborts.

F.ERRS: BYTE; 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; only the values Ø through 15 are used. If you attempt a file operation other than CLOSE or DISCARD and F.ERRS is not zero, the program immediately aborts if F.TRAP is FALSE. However, if F.TRAP is TRUE, the attempted file operation is ignored and the program continues.

CLOSE and DISCARD do not examine the initial value of F.ERRS, so they 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 can change the buffer variable or READ procedure input variables. (See Appendix A, "Compiler Error Messages," for a complete listing of error messages and warnings.)

Also at the **extend level**, you can 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.

At the **extend level**, you can also 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. The interface for the FCBFQQ type (and any other types needed) is 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.

8 REFERENCE AND OTHER TYPES

The array, record, and set types discussed in Section 6 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 Section 7, "Files," is a structure that varies in size but whose form and means of access are predetermined.

This section discusses 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.

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.

Our version of 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. Programs which use pointers are portable.

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.

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 can perform only the following actions on pointers:

- o assign them
- o test them for equality and inequality with the two operators = and <>
- o pass them as value or reference parameters
- o dereference them with the caret(^), or at the **extend level** use the question mark (?) or the at sign (@)

In declarations of pointer types, the caret (^) prefixes the type pointed to; in program statements, it dereferences a pointer so that the value pointed to can be assigned or operated on.

Example:

```
VAR P : ^WORD;
  BEGIN NEW(P); {Allocate a word on the heap}
                {and make P point to it}
        P^ := 1 {Assign to that word}
  END;
```

Every pointer type includes the pointer value NIL. NIL pointer value if the pointer does not point to anything.

Pointers are frequently used to create list structures of records, as shown in the following example, where TREETIP is the pointer type with reference type TREE:

```
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, so that pointers can point to themselves and to records of which they themselves are fields.

The declaration for a pointer type 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 an 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. The compiler 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 reference 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.

Our version of 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 END;  {This is illegal!}
```

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 could not 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.

Pointers are implemented by this version of Pascal as relative addresses in the data segment.

If the initialization-checking switch ($INITCK) 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 that have been referenced to a heap item that has been DISPOSEd, or a value that is not valid as a heap reference.

ADDRESS TYPES

As a system implementation language, 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, referred to by the keyword ADR. A relative address is a 16-bit offset into the default data segment. This takes two bytes of storage.

- Segmented, referred to by the keyword ADS. A segmented address is a 16-bit offset and a 16-bit segment. This takes four bytes of storage.

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 caret to dereference}
  {the address types.}
  {Output is 1 and 3.1415.}
END.
```

In this version of Pascal, you can 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
```

That is, a variable of type ADR can be used as a variable of type

```
RECORD
  R: WORD
  END;
```

and a variable of type ADS can be used as a variable of type

```
RECORD
  R:WORD;
  S: WORD
  END;
```

(although, of course, ADR and ADS types are not equivalent to the above RECORD types.)

You can specify the assigned value in hexadecimal notation. You can also assign to a segment field with the ADS type using the field notation .S (segment address). Thus, you can 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;
```

This version of Pascal supports these two predeclared address types:

```
ADRMEM = ADR OF ARRAY [0..32765] OF BYTE;
ADSMEM = ADS OF ARRAY [0..32765] 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 can 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 (Ø, 32)^;
  {Get word at address 32:Ø}
  W := (ADS W).S;
  {Get value of DS 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 functions that yield addresses cannot currently be used as the target of an assignment (or as a reference parameter or WITH record), as shown:

```
VAR AW : ADSWORD;
CONST ADSCON = ADSWORD (256, 64);    {OK}
FUNCTION SOME_ADDRESS: ADSWORD;      {OK}
BEGIN
  ADSWORD (Ø, 32)^ := W; {NOT PERMITTED}
  ADSCON^ := 12;         {NOT PERMITTED}
  SOME_ADDRESS^ := 1ØØ;  {NOT PERMITTED}
  AW^ := W; {Permitted.  AW is neither a}
            {constant nor function call}
END;
```

Reference Parameters

You can pass the segmented address of a variable (ADS) the same way that you can pass a relative address, by using either of the keywords VARS or CONSTS as a parameter prefix (instead of VAR and CONST).

If P is of type ADS of FOO, then P^ can be passed to a VARS formal parameter, such as VARS X: FOO. However, it cannot be passed to a VAR formal parameter.

When only relative addresses are specified, the default data segment is assumed. This applies to ADR variables and VAR parameters.

A VAR parameter is passed as the default data segment offset of a variable. For a VARS parameter both the segment address and the offset value are passed.

Both VARS parameters and ADS variables have the offset (.R) value in the WORD with the lower address and the segment (.S) value at the higher address.

USING THE ADDRESS TYPES

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

The caret (^) has a higher precedence than the unary operators ADR or ADS. Because the caret (^) selector can appear after any address variable to produce a new variable, it can occur, for example, in the target of an assignment, in 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, constant, or expression.

Pascal is a strongly typed language; two pointer variables are compatible only if they have the same type, not if they only point to the same type. However, two address types are considered the same type if they are both either ADR or ADS types. For example, you can 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 of type ADR OF STRING (200) and P2 is of any ADR OF type, the assignment P1^ := 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.

Within limits, you can combine and intermingle the two address types. The following example illustrates the rules that apply:

```
VAR
  P: ADS OF DATA;
  {P is segmented address of some 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 address of 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 the subsection, "Address Types."

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 a variable (pointer) to another variable (reference.) 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; and the address type is generally not portable. Address types are portable only if you restrict yourself to using ADS and never assign to fields .R and .S.

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

- 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.

- 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.

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 this version of Pascal, some limitations on the use of PACKED structures currently apply:

- In sets, files, and arrays of characters, the prefix PACKED is always ignored, except for type checking, and 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.

- 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.

- 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 the subsection, "Arrays" in Section 6, "Arrays, Records, and Sets."

PROCEDURAL AND FUNCTIONAL TYPES

Procedural and functional types are different from other Pascal types. (Wherever the term "procedural" is used from hereon, both procedural and functional is implied.) You cannot declare an identifier for a procedural type in a TYPE section; nor can you declare a variable of a procedural type. However, you can 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 the 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 this version of 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 the subsection "Procedural and Functional Parameters" in Section 13, "Introduction to Procedures and Functions," for more information about procedural types.

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 and your birth date.

You can give a constant 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:

- Numeric constants are one of the several number types: REAL8, INTEGER, WORD, or INTEGER4.
- Character constants are strings of characters enclosed in single quotation marks and are called string literals in this version of Pascal.
- Structured constants, available at the **extend level**, include constant arrays, records, and typed sets.

Constant expressions, also available at the **extend level**, let you compute a constant based on the values of previously declared constants in expressions.

Numeric constants, character constants, structured constants, and constant expressions are each discussed in a subsection below.

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 (for example, ORD (BLUE)).

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 cannot redefine it. Also, the null set is a constant of any set type.

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

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 four statements that identify constants (beginning after the word "CONST"):

```
PROGRAM DEMO (INPUT, OUTPUT);
CONST DAYSINYEAR = 365;
      DAYSINWEEK = 7;
      NAMEOFPLANET = 'EARTH';
      WORKDAYS = DAYSINWEEK - 2;
```

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;
```

Our version of Pascal, however, 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);
{Structured constant of type COMPLEX}
VAR   PIX : COMPLEX;
TYPE  IVEC = ARRAY [1..3] OF COMPLEX;
CONST PIVEC = IVEC (PII, PII, COMPLEX (0.0,
                1.0));
```

NUMERIC CONSTANTS

Numeric constants are irreducible numbers such as 45, 12.3, and 9E12. The notation of a numeric constant generally indicates its type: REAL8, INTEGER, WORD, or INTEGER4.

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

```
ALPHA := +1Ø          {is legal}

ALPHA + -1Ø           {is illegal}
```

Blanks embedded within constants are not permitted.

The compiler truncates any identifier that exceeds a maximum of 31 characters and gives a warning when this occurs.

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                  Ø.17
+12.345               ØØ7
-1.7E-1Ø             -26.Ø
 17E+3                26.ØE12
-17E3                 1E1
```

Numeric constants can appear in any of the following:

- CONST sections
- expressions
- type clauses
- set constants
- structured constants
- CASE statement CASE constants
- variant record tag values

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

REAL CONSTANTS

The type of a number is REAL if the number includes a decimal point or exponent. Real numbers use the IEEE format. For REAL4 values, the range is barely seven significant digits (24 bits), with an exponent range of E-38 to E+38. REAL8 values have a range of over fifteen significant digits (53 bits) and an exponent range of E-3Ø6 to E+3Ø6.

There is, however, a distinction between REAL values and REAL constants. In IEEE format, REAL numeric constants are kept in double precision and so can range from about 1E-3Ø6 to 1E3Ø6.

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 can 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 lowercase "e" are allowed in REAL numbers. "D" and "d" are also allowed to indicate an exponent. This provides compatibility with other languages.

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.)

INTEGER, WORD, AND INTEGER4 CONSTANTS

The type of a non-REAL numeric constant is INTEGER, WORD, or INTEGER4. Table 9-1 shows the range of values that constants of each of these types can assume.

Table 9-1. INTEGER, WORD, and INTEGER4 Constants.

Type	Range of Values (minimum/maximum)	Predeclared Constant
INTEGER	-MAXINT to MAXINT	MAXINT=32767
WORD	Ø to MAXWORD	MAXWORD=65535
INTEGER4	-MAXINT4 to MAXINT4	MAXINT4=2147483547

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

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

- A constant identifier from -MAXINT to MAXINT becomes an INTEGER.
- A constant identifier from MAXINT+1 to MAXWORD becomes a WORD.
- 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 necessary; 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(Ø)..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 9-2.

At the standard level, any numeric constant (that is, a literal or identifier) can have a plus (+) or minus (-) sign.

Table 9-2. Constant Conversions.

Constant	Assumed Type
0	INTEGER could become WORD or INTEGER4
-32768	INTEGER4 only
32768	WORD could become INTEGER 4
0..20000	INTEGER subrange
0..50000	WORD subrange
0..80000	Invalid: no INTEGER4 subranges
-1..50000	Invalid: becomes 65535..50000 (that is, -1 is treated as 65536)

NONDECIMAL NUMBERING

At the **extend level**, our version of Pascal supports not only decimal 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, that is, the number preceding it is the radix of the number following the sign.

Examples of numbers in nondecimal notation:

```
16#FFØ2     hexadecimal
1Ø#987      decimal
 8#776      octal
 2#111ØØ    binary
```

Leading zeros are recognized in the radix, so a number like ØØ8#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 can be used for INTEGER, WORD, or INTEGER4 constants. You must not use nondecimal notation for REAL8 constants or numeric statement labels.

CHARACTER STRINGS

Most Pascal manuals refer to sequences of characters enclosed in single quotation marks as string. In this version of Pascal, they are called string literals to distinguish them from string constants, which can 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 this version of 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, that is, STRING (1), if necessary. For example, a constant ('A') of type CHAR could be assigned to a variable of type STRING (1).

A string literal must fit on a line.

String literals are usually enclosed in single quotation marks. 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.

A literal apostrophe (single quotation mark) is represented by two adjacent single quotation marks (for example, 'DON''T GO').

The null string ('') is not permitted.

The constant expression feature (discussed in the subsection, "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(n). This is useful for representing string constants that are longer than a line or that contain nonprinting characters. For example:

```
'UNDERLINED TEXT' * CHR(13)
                  * STRING(DO 15 OF '_')
```

The last string in the example above is a structured constant. Structured constants are discussed in the subsection below.

The LSTRING feature of this version of Pascal adds the super array type LSTRING. LSTRING is similar to PACKED ARRAY [Ø..n] OF CHAR, except that element Ø contains the length of the string, which can vary from Ø to a maximum of 255. If necessary, a constant of type STRING (n) or CHAR changes automatically to type LSTRING. (See the subsection, "LSTRINGs," in Section 6, "Arrays, Records, and Sets.")

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

Examples of string literal declarations:

```
CONST

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

STRUCTURED CONSTANTS

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

At the **extend level**, however, you can 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.

- 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;
```

- 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;
```

- 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 the subsection, "Super Arrays," in Section 6, "Arrays, Records and Sets.") 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".

Set constant expressions, such as ['_'] + LETTERS, or file constant expressions are not supported. 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 the subsection, "Procedural and Functional Parameters" in Section 13, "Introduction to Procedures and Functions.")

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 can use either in an expression or to define the value of a constant identifier. Set constants with an explicit type can 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.

CONSTANT EXPRESSIONS

Constant expressions in our version of 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 can contain only constants that you have declared earlier, the following is illegal:

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

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

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

Table 9-3 shows the functions and operators you can use with REAL, INTEGER, WORD, and other ordinal constants, such as enumerated and subrange constants.

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 9-3 for numeric constants, you can 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'}
```

Note that the asterisk (*) works only with string constants and not with variables. For variable strings you must use the procedure CONCAT.

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.

Table 9-3. Constant Operators and Functions.

Type of Operand	Operators and Functions
REAL, INTEGER	Unary plus (+) Unary minus (-)
INTEGER, WORD	+ DIV OR HYBYTE() - MOD NOT LOBYTE() * AND XOR BYWORD()
Ordinal types	< <= CHR() LOWER() > >= ORD9Ø UPPER() = <> WRD()
Boolean	AND NOT OR
ARRAY	LOWER() UPPER()
Any type	SIZEOF() RETYPE()

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 can 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 can be denoted by array indexes or record fields or the dereferencing of pointer or address variables.

The compiler itself can sometimes create "hidden" variables, allocated on the stack, in circumstances such as the following:

- When you call a function that returns a structured result, the compiler allocates a variable in the caller for the result.

- When you need the address of an expression (for example, 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.

- The initial and final values of a FOR loop can require allocating a variable.

- When the compiler evaluates an expression, it can allocate a variable to store intermediate results.

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

DECLARING A VARIABLE

A variable declaration consists of the identifier for the new variable, followed by a colon and a type. You can 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
```

A variable must be declared if it is used in a statement.

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

- VAR section of a program, procedure, function, module, interface, or implementation
- formal parameter list of a procedure, function, or procedural parameter

In a VAR section, you can declare a variable to be of any legal type; in a formal parameter list, you can include only a type identifier (that is, you cannot 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.}

TYPE CLRS = ARRAY [1..10] OF COLOR;
PROCEDURE NAME (GEORGE : CLRS);
{Legal; CLRS is a type identifier.}
```

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 the subsections "The Buffer Variable" and "Extend Level I/O" in Section 7, "Files," for further information on buffer variables and FCBFQQ fields, respectively.

THE VALUE SECTION

You use the VALUE section to give initial values to variables in a program, module, procedure, or function. You can also initialize the variable in an implementation, but not in an interface. (See Section 16, "Compilable Parts of a Program," for information on implementations and interfaces.)

The VALUE section can include only statically allocated variables, that is, any variable declared at the module, program, 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 can 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 cannot assign a variable to another variable. The last line in the following example is illegal, since "I" must be a constant:

```
CONTS 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.

USING VARIABLES AND VALUES

At the standard level, denotation of a variable can designate one of three things:

- an entire variable
- a component of a variable
- a variable referenced by a pointer

The value assigned to a variable can be any of the following:

- a variable
- a constant
- a function designator
- a component of a value
- a variable referenced by a reference value

In an assignment statement, the left-hand side denotes a variable (or a component of one), and the right hand side denotes a value.

At the **extend level**, a function can also return an array, record, or set. The same syntax used for variables can 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 can 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 can declare constants of a structured type. Components of a structured constant use the same syntax as variables of the same type. (See the subsection "Constant Expressions" in Section 9, "Constants" 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];
{that is, 3.14 * 9.42}
Y := REAL3 (1.1, 2.2, 3.3) [2];
{that is, 2.2}
```

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).

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. (Compatibility is discussed in the subsection "Type Compatibility" in Section 4, "Introduction to Data Types.") 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 element of the array}
{corresponding to the index C.}

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

```
BETAMAX [12] [-3]
BETAMAX [12,-3]
{These two are equivalent.}

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

- with the notation LSTR [Ø], to access the length as a CHAR component
- with the notation LSTR.LEN, to access the length as a BYTE value

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. A period (.) separates the fields. In a WITH statement, you give the record variable or value only once. Within the WITH statement, you can use the field identifier of a record variable directly.

Examples of field variables and values:

```
PERSON.NAME := 'PETE';

PEOPLE.DRIVER.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.

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, obtaining its value can involve first reading the value from the file. (This is called lazy evaluation; see the subsection "Lazy Evaluation" in Section 15, "File-Oriented Procedures and Functions," 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, after the position of the file is changed with a GET or PUT procedure, the value of the buffer variable could be incorrect.

Examples of file reference variables:

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

REFERENCE VARIABLES

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

- pointer variables and values
- ADR variables and values
- 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 a caret (^) 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 can also append a caret (^) to a function designator for a function that returns a pointer or address type. In this case, the caret 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 caret}
{dereferences the reference value returned,}
{assigning the referenced data to DATA1}

DATA2 := FUNK2 (K, L)^.FOO [2]
{FUNK2 returns a reference value.  The caret}
{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
```

ATTRIBUTES

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

Table 10-1 displays the attributes provided for variables. Each of the variable attributes is discussed in detail in the subsections below.

Table 10-1. 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.

EXTERN, PUBLIC, and ORIGIN also apply to procedures and functions. (EXTERN is a procedure and function directive; PUBLIC and ORIGIN are procedure and function attributes. See the subsection "Directives and Attributes" in Section 13, "Introduction to Procedures and Functions," for a discussion of procedure and function attributes and directives.)

You can 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).

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.}
```

The brackets can occur in either of two places:

- An attribute in brackets after a variable identifier in a VAR section applies to that variable only.

- An attribute in brackets after the reserved word VAR applies to all of the variables in the section.

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.

Example of STATIC variable declarations:

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

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

THE PUBLIC AND EXTERN ATTRIBUTES

The PUBLIC attribute indicates a variable that can be accessed by other modules; the EXTERN attribute identifies a variable that resides in some other module.

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 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 modules that}
{declare BASE_PAGE with the EXTERN attribute.}
```

Memory for PUBLIC variables is usually allocated by the compiler, unless you also give them an ORIGIN. Giving a variable both the PUBLIC and ORIGIN attributes tells the compiler that a global name has an absolute address.

Note that if a variable is declared PUBLIC then the identifier is kept in the symbol table produced by the Linker and can be accessed symbolically when you are using the Debugger.

Note that this does not refer to Pascal error handling routines, but to the Debugger available with the standard software for your workstation.

PUBLIC cannot be combined with PORT.

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

Memory for EXTERN variables is 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.

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. ORIGIN and PORT are actually implemented in the same way. The PORT attribute is included for compatibility with other Pascals. 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 PORT variable declarations:

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

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.

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 can 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.

Giving the PORT or 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 Ø] FIRST, SECOND:  BYTE;
{ILLEGAL!}
```

THE READONLY ATTRIBUTE

The READONLY attribute prevents assignments to a variable. It also prevents the variable being passed as a VAR or VARS parameter. A READONLY variable cannot be read with a READ statement or used as a FOR control variable. (READ is discussed in Section 15, "File-Oriented Procedures and Functions," and FOR is discussed in Section 12, "Statements.") You can 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.

COMBINING ATTRIBUTES

You can 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:

- If you give a variable the EXTERN attribute, you cannot give it the PORT, ORIGIN, or PUBLIC attribute in the current compiland.
- If you give a variable the PORT attribute, you cannot give it the ORIGIN, PUBLIC, or EXTERN attribute at all.
- If you give a variable the ORIGIN attribute, you cannot also give it the PORT or EXTERN attribute. However, you can combine ORIGIN with PUBLIC.
- If you give a variable the PUBLIC attribute, you cannot also give it the PORT or EXTERN attribute. However, you can combine PUBLIC with ORIGIN.
- You can use STATIC and READONLY with any other attribute.

11 EXPRESSIONS

Expressions are constructions that evaluate to values. Table 11-1 illustrates a variety of expressions, which, if A = 1 and B = 2, evaluate to the value shown.

Table 11-1. Expressions.

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

The operands in an expression can 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 as complicated as you wish.

The available operators, in the order in which they are applied, are listed below. **Extend level** operators are shown in boldface text.

1. Unary NOT **ADR ADS**
2. Multiplying * / DIV MOD AND
3. Adding + - OR **XOR**
4. Relational = <> <= >= < > IN

A 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 operators only apply to the following types:

INTEGER	INTEGER4
WORD	BOOLEAN
REAL	SET

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

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.

> NOTE
>
> 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:

- from INTEGER to REAL or INTEGER4
- from WORD to INTEGER4

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

+ - *

These operators operate on 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 result type is the type of the operands. If either operand is REAL8, the result type is REAL8; if either is REAL4 and none REAL8, the result is REAL4. If either operand is INTEGER4, the result type is INTEGER4.

Unary plus (+) and minus (-) are supported, along with the binary forms. Unary minus on a WORD type is 2's complement (NOT is 1'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 illegal. For the same reason, (-256 AND X) is interpreted as -(256 AND X).

This is a "true" division operator. The result is always REAL. Operands can be INTEGER, INTEGER4, REAL, or REAL8 (not WORD.)

Examples of division:

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

DIV MOD

These are the operators for integer division 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 results 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.

Our version of Pascal differs from the 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.

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 ones 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 ($INITCK) 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
```

BOOLEAN EXPRESSIONS

The Boolean operators at the standard level are:

NOT	AND	OR
=	<	>
<>	<=	>=

XOR is available at the **extend level.**

You can also use P <> Q as an exclusive OR function. Since FALSE < TRUE, P <= Q denotes the Boolean operation "P implies Q."

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 do not:

```
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 run-time error. This evaluation may or may not take place. Sometimes both operands are evaluated during optimization, and sometimes the evaluation of one can cause the evaluation of the other to be skipped. In the latter case, either operand can be evaluated first.

Instead, use the following construction:

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

(See the subsection "Sequential Control" in Section 12, "Statements," 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>;
```

(See Section 8 "Reference and Other Types" for a discussion of the address type.)

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 can 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 current length of the other and all characters up to the current 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. (See the subsection "Using STRINGs and LSTRINGs" in Section 6, "Arrays, Records, and Sets," for more information.)

The six relational operators =, <>, <=, >=, <, and > have their normal meaning when applied to numeric, enumerated, CHAR, or string operands. The subsection, "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 logical AND and OR, the following equivalent statements are incorrect:

```
IF I < 1Ø AND J = K THEN
IF I < (1Ø AND J) = K THEN
```

Instead, you must write:

```
IF (I < 1Ø) AND (J = K) THEN
```

Also, you cannot 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 <> Ø THEN
```

Note that the following is also not allowed, if I is not an INTEGER constant:

```
$IF I $THEN
```

The inclusion of special "not-a-number" (NaN) 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:

- A < B is false.
- A <= B is false.
- A > B is false.
- A >= B is false.
- A = B is false.
- 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.

SET EXPRESSIONS

Table 11-2 shows the operators that apply differently to sets than to other types of expressions.

Table 11-2. 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 of some type T, is treated as if it were SET OF T. (T is restricted to the range from Ø to 255 or the equivalent ORD values.) Unless one operand is a constant or constructed set, both operands must be either PACKED or not PACKED.

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).

Operations on sets with up to 16 elements generate inline code.

Angle brackets <> are set operators only at the **extend level**, since the ISO standard does not support them for sets. They test that a set is a

proper subset or superset of another set. A set can be a subset but is not a proper subset of itself.

Expressions involving sets can 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 a 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.

Operations on sets use the stack instead of the heap for temporaries.

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 function designator.}
```

The 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 the subsection, "Procedure and Function Parameters," in Section 13, "Introduction to Procedures and Functions" 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:

- 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 does nothing else (such as assign to a static variable or do input/output), it is called a pure function.

- The second type of function can 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 can return either a simple type or a pointer. At the **extend level**, a function can return any assignable type (that is, any type except a file or super array).

At the standard level, a pointer that is a function designator (that is, returned by a function) can only be compared, assigned, 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 the subsection "Using Variables and Values" in Section 10, "Variables and Values," for 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 NAD 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.

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 can occasionally give you unexpected overflow errors. For example,

```
(I - 100) + (J - 100)
```

are optimized into the following:

```
(I + J) - 200
```

This can result in an overflow error, although the original expression did not (for example, if "I" and "J" were each 16400).

This optimization can be suppressed by turning on the $DEBUG switch (except for some constant folding, for example replacing an expression such as 3*6 by 18). The $SIMPLE metacommand does not suppress it.

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 can be different due to precision loss. Common subexpressions, such as 2 * X in SIN (2 * X) * COS (2 * X), can still be calculated just once and reloaded as necessary, but they are saved in a special 80-bit intermediate precision.

The order of evaluation can be fixed by parentheses:

```
(A + B) + C
```

is evaluated by adding A and B first, but

```
A + B + C
```

can 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 CONST or CONSTS 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 <> Ø THEN
    SUM := SUM (SUM, (SUM (B, Ø) - 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. 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 while putting the function in parentheses, as in (F(X)), forces evaluation of the function.

OTHER FEATURES OF EXPRESSIONS

EVAL and RESULT are two procedures available at the **extend level** for use with expressions. EVAL uses a function as a procedure; RESULT yields the current value of a function within a function or nested procedure or function.

At the **extend level**, the function RETYPE allows you to change the type of a value.

THE EVAL PROCEDURE

EVAL evaluates its parameters without actually calling anything. Generally, you use EVAL to use a function as a procedure. 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)
```

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, while 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 < Ø THEN ABSVAL := -RESULT (ABSVAL)
END;
```

THE RETYPE FUNCTION

Occasionally, you need to change the type of a value. You can do this with the RETYPE function, available at the **extend level**. If the new type is a structure, RETYPE can be followed by the usual selection syntax.

> NOTE
>
> You must use RETYPE with caution. It uses a complicated algorithm and is sometimes unpredictable.

Examples of the RETYPE function:

```
RETYPE (COLOR, 3)             {inverse of ORD}
RETYPE (STRING2, I*J+K) [2]   {effect can vary}
```

12 STATEMENTS

The body of a program, procedure, or function contains statements. Statements denote actions that the program can execute.

This section 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 12-1 lists the statements in each category.

Table 12-1. Statements.

Simple	Structured
Assignment (:=)	Compound
Procedure	IF/THEN/ELSE
GOTO	CASE
BREAK	FOR
CYCLE	WHILE
RETURN	REPEAT
Empty	WITH

SYNTAX

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 elements of statement syntax are discussed in the following sections.

SEPARATING STATEMENTS

Semicolons separate statements, rather than terminate them. However, since our version of 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, "executes" an empty statement ten times, then executes the array assignments once. Since there are occasional legitimate uses for repeating an empty statement, no warning is given when this occurs. (The FOR statement is discussed further on in this section.)

The semicolon also follows the reserved word END at the close of a block of program statements.

THE RESERVED WORDS BEGIN AND END

Whenever you want a program to execute a group of statements, instead of a single simple statement, you can enclose the block with the reserved words BEGIN and END. Follow END with a semicolon.

For example, the following group of statements between BEGIN and END are all 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')
```

Note that a semicolon is not necessary after the last statement within the BEGIN..END block. Since the empty statement is legal in this version of Pascal, however, you can use a semicolon after the last statement without causing a warning or execution error.

At the **extend level**, you can substitute a pair of square brackets for the pair of keywords BEGIN and END.

```
IF (MAX > 10) THEN
[ MAX = 10;
  MIN = 0;
  WRITELN (MAX,MIN)];
WRITELN ('done')
```

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.

When both a CASE constant list and a GOTO label precede a statement, 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 GO TO 123
```

SIMPLE STATEMENTS

A simple statement is one in which no part constitutes another statement. Simple statements in standard Pascal are the following:

- o assignment statement
- o procedure statement
- o GOTO statement
- o 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** adds three simple statements: BREAK, CYCLE, and RETURN.

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 a colon together with an equal sign (:=).

Examples of assignment statements:

```
A := B;
{A is assigned the value of 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 can involve indexing an array or dereferencing a pointer or address. If

it does, the compiler can, 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.

Note that an assignment to a nonlocal variable (including a function return) causes the compiler to put an equal sign (=) or percent sign (%) in the G column of the listing file. (See the subsection, "Listing File Format," in Section 18, "Using the Compiler," 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 can occur either within the actual body of the function or in the body of a procedure or function nested within it. (Using functions is discussed in detail in Section 13, "Introduction to Procedures and Functions.")

If the range-checking switch ($RANGECK) is on, an assignment to a set, subrange, or LSTRING variable can imply a run-time call to the error checking code.

Each section of code without a label or other point that could receive control is eligible for rearrangement and common subexpression elimination by the optimizer. 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:

- Get the value of A and save it.
- Add the value of B and save the result.
- Add the value of C and assign it to X.
- Assign the saved A + B value to Y.
- 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 can 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 can limit the ability of the optimizer to find common subexpressions:

- assignment to a nonlocal variable
- assignment to a reference parameter
- assignment to the referent of a pointer
- assignment to the referent of an address variable
- calling a procedure
- 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.

PROCEDURE STATEMENTS

A procedure statement executes the procedure denoted by the procedure identifier. (Using procedures is discussed in detail in Section 13, "Introduction to Procedures and Functions.)

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.

This version of Pascal includes a large number of predeclared procedures. See Section 14, "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 the subsection "Parameters to Procedures and Functions" in Section 13, "Introduction to Procedures and Functions.")

GOTO STATEMENTS

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:

- 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.

- 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 can 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 in to 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;
    {That was 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 SINGLE DO
          3: GOTO 2
          {OK to jump out of loops.}
    UNTIL DATE;
    1: GOTO 11;
    {Illegal.  Cannot jump into procedure from}
    {program.}

END.
```

STRUCTURED STATEMENTS

Structured statements are themselves composed of other statements. There are four kinds of structured statements:

- compound statements
- conditional statements
- repetitive statements
- WITH statement

The control level for statements is shown in the C (control) column of the listing file. (See the subsection "Listing File Format" in Section 17, "Metacommands," for more information.)

COMPOUND STATEMENTS

The compound statement is a sequence of 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 conditional and repetitive control structures (except REPEAT) used in this version of Pascal operate on a single statement, not on multiple statements with ending delimiters. If you want those structures to operate on a sequence of several statements, you can make a compound statement out of the sequence by enclosing it with the reserved words BEGIN..END.

In this context, BEGIN and END serve as punctuation, like semicolon, colon, or parentheses. If you prefer, at the **extend level** you can 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, because END has other uses than forming compound statements.

Brackets cannot be used, however, 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 := Ø];

  WHILE P.N <> NIL DO
    [Q := P;  P := P.N;  DISPOSE (Q)];

  FUNCTION R2 (R: REAL): REAL;
    [R2 := R * 2]
    {Illegal.}
```

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.

The IF Statement

The IF statement can take one of the two forms:

- IF <expression> THEN <statement1>
- IF <expression> THEN <statement1> ELSE <statement2>

where

<expression>
: is any Boolean expression.

<statement1>
: can be any statement.

<statement2>
: can be any statement.

IF, THEN, and ELSE are reserved words. No semicolon can precede the reserved word ELSE.

The <expression> is evaluated first. If its value is TRUE, <statement1> is executed. If the value is FALSE,

- In the first case above nothing is done, control passes to the next statement.

- In the second case <statement2> is executed.

Examples of IF statements:

```
IF I > Ø THEN I := I - 1
{No semicolon between IF and ELSE.}
ELSE I := I + 1;

IF (I <= TOP) AND (ARRI [I] <> TARGET) THEN
    I := I + 1;
```

<statement1> can be any one statement. To specify a sequence of statements in the THEN clause above, make them into one compound statement (by enclosing them in a BEGIN..END pair.) The same applies to <statement2>.

Example:

```
IF I > Ø
THEN
  BEGIN
    J := 3;
    K := 4
  END

ELSE
  BEGIN
    J := 7;
    K := 8
  END;
```

<statement1> and <statement2> can themselves be conditional statements. In the case of such a "nested" conditional statement, an ELSE is paired with the inner IF clause, as in the following example:

```
IF I>Ø
THEN
  IF InnerI > Ø
  THEN J := 5
  ELSE J := 6;
```

The ELSE is paired with IF InnerI > Ø, so that the above is equivalent to

```
IF I>Ø
THEN
  BEGIN
    IF InnerI >Ø
    THEN J := 5
    ELSE J := 6
END
```

and not to

```
IF I>Ø
THEN
  BEGIN
    IF InnerI >Ø
    THEN J := 5
  END
ELSE J := 6
```

The following are additional examples of how you can use the IF statement:

```
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.}
```

The Boolean expression following an IF can include the sequential control operators described in the subsection "Sequential Control" at the end of this section.

The CASE Statement

The CASE statement is similar to the conditional statement in that it specifies that only one (or none) of a number of statements must be executed.

CASE and OTHERWISE are reserved words.

The syntax of the CASE statement is:

```
CASE <index> OF
  <value1>: <statement1>;
  <value2>: <statement2>;
  ..
  <valueN>: <statementN>
END
```

where

<index>
: is any expression of an ordinal type.

<value1>
: can be any constant of the same type, or a list of constants separated by commas, or a subrange of the same type. The same applies to <value2>,..<valueN>.

  Each constant in the type can be defined by not more than one <value>.

<statement1>
: can be any one statement. (A compound statement can be used if you want more than one statement there). It can also be another conditional and case statement. This applies also to <statement2>, ... <statementN>, <statement> in the example below.

At the **extend level**, the CASE statement can also look as follows:

```
CASE <index> OF
  <value1>: <statement1>;
  <value2>: <statement2>;
  ..
  <valueN>: <statementN>;
  OTHERWISE <statement>
END
```

When the CASE statement is executed, the <index> is evaluated. If its value is equal to <value1> (or, if <value1> is a list of constants or a subrange and <index> equals one of the constants specified by <value1>), then <statement1> is executed; the rest of the statements are ignored. Control then passes to the statement following the CASE statement.

If the <index> value is equal to a constant defined by <value2>, only <statement2> is executed. (The same is true for the rest of the <value>'s.)

If <index> is not equal to any of the constants defined by the <value>'s, then one of the following occurs:

- If the OTHERWISE clause is present, <statement> is executed, and control is passed to the statement following the CASE statement.

- If there is no OTHERWISE clause, none of the <statement>'s is executed, and control is passed to the statement following the CASE statement.

Example:

```
VAR OPERATOR (PLUS,MINUS,TIMES);
    NEXTCH : CHAR;
BEGIN
  .
  .
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', '_'          : WRITE ('Identifier');
  '+', '-', '*', '/'     : WRITE ('Operator');
  {Commas separate CASE constants}
  {and ranges of CASE constants.}
  OTHERWISE
    WRITE ('Unknown Character')
    {that is, if any other character}
END
```

Note that <index> cannot be an INTEGER4, since INTEGER4 is not an ordinal type.

The CASE syntax for <valuel>..<valueN> 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 can 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:

- If the range-checking switch ($RANGECK) is on, a run-time error is generated.
- If the range-checking switch is off, the result is undefined.

> NOTE
>
> It is not recommended to use the CASE statement without the $RANGECK metacommand on, unless you include an OTHERWISE clause in the statement.
>
> Depending on optimization, the code generated by the compiler for a CASE statement can 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.

You can include an empty OTHERWISE clause to force control to pass on to the next statement, if you wish.

A semicolon (;) can 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.

REPETITIVE STATEMENTS

Repetitive statements specify repeated execution of a statement. In standard Pascal, these include the WHILE, REPEAT, and FOR statements.

At the **extend level**, 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.

The WHILE Statement

The WHILE statement repeats a statement zero or more times, until a Boolean expression becomes false. The syntax of the WHILE statement is

```
WHILE <expression> DO <statement>
```

where

<expression>
: is any Boolean expression.

<statement>
: is any statement.

WHILE and DO are reserved words.

<statement> is executed while <expression> is TRUE, that is:

1. <expression> is evaluated.

2. If its value is FALSE, the execution of the WHILE statement is terminated and control passes to the next executable statement.

   If its value is TRUE, <statement> is executed and control returns to step 1.

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 can include the sequential control operators described in the subsection, "Sequential Control."

Use WHILE if it is possible that no iterations of the loop will be necessary; use REPEAT where you expect that at least one iteration of the loop will be required.

The REPEAT Statement

The REPEAT statement repeats a sequence of statements one or more times, until a Boolean expression becomes true.

The syntax of the REPEAT statement is

```
REPEAT
<statement1>;
.
.
<statementN>
UNTIL <expression>
```

where

<statement1>..<statementN>
: are any statements.

<expression>
: is any Boolean expression.

REPEAT and UNTIL are reserved words.

The REPEAT statement is executed as follows:

1. <statement1> through <statementN> are executed.

2. <expression> is evaluated.

   If its value is FALSE, control returns to Step 1.

   If its value is TRUE, control passes to the statement following the REPEAT statement, that is, the loop terminates.

Examples of REPEAT statements:

```
REPEAT
  READ (LINEBUFF);
  COUNT := COUNT + 1
UNTIL EOF;

REPEAT GAME UNTIL TIRED;
```

The Boolean expression in a REPEAT statement can include the sequential control operators described in the subsection, "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.

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.

INTEGER4 values cannot be used to control FOR statements.

The syntax of the FOR statement can be one of the following:

- FOR <control variable> := <initial value> TO <final value>
  DO <statement>

- FOR <control variable> := <initial value> DOWNTO <final value>
  DO <statement>

  where

  <control variable>
  is any variable of any ordinal type (cannot be an INTEGER4).

  <initial value> and <final value>
  are expressions compatible with the type of <control variable>.

  <statement>
  is any statement.

  FOR, TO, DOWNTO and DO are reserved words.

The FOR statement is executed as follows:

1. <initial value> is evaluated. Then <final value> is evaluated.

2. <control variable> is assigned the value of <initial variable>.

3. <control variable> is compared with <final value>.

   The test is made if <control variable> is

   - less than or equal to <final value> if TO is used.

   - greater than or equal to <final value> if DOWNTO is used.

   If the test does not pass, the execution of the FOR statement is terminated and control passes to the next statement.

If the test passes, <statement> is executed. Then <control variable> is

- incremented if TO is used, and control passes to step 3, above, or

- decremented if DOWNTO is used, and control passes to step 3, above

Examples:

```
FOR I := 1 TO 1Ø DO
{I is the control variable.}
  SUM := SUM + VICTORVECTOR [I]

FOR CH := 'Z' DOWNTO 'A' DO
{CH is the control variable.}
  WRITE (CH);
```

You can 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 following ISO standard rules apply to the control variable in FOR statements:

- It must be of an ordinal type.

- It must also be an entire variable, not a component of a structure.

- 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**, the control variable can 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 detected by this compiler.

- 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.

- 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

- The initial value is greater than the final value in the TO case.

- 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. (See Section 14, "Available Procedures and Functions," for a description of PRED and SUCC.) The value of the control variable, after a FOR statement terminates naturally (whether or not the body executes), is undefined. It can vary due to optimization and, if $INITCK is on, can 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.

A control variable with the STATIC attribute can also be more efficient than one that is not.

At the **extend level**, you can use temporary control variables:

```
FOR VAR <control-variable>
```

The prefix VAR causes the control variable to be declared local to the FOR statement (that is, 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;
```

The BREAK and CYCLE Statements

At the **extend level**, the BREAK and CYCLE statements are allowed in addition to the simple statements already described. Both statements are functionally equivalent to a GOTO statement. In theory, a program using the **extend level** BREAK and CYCLE statements does not need to use any GOTO statements.

These statements perform the following functions:

- BREAK exits the currently executing loop.

  A BREAK statement is a GOTO to the first statement after a repetitive statement.

- CYCLE exits the current iteration of a loop and starts the next iteration.

A CYCLE statement is a GOTO to an implied empty statement after the body of a repetitive statement. Thus 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.

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 do not 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;
```

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 can 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;
```

THE RETURN STATEMENT

At the **extend level** the RETURN statement exits the current procedure, function, program, or implementation.

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.

Examples:

```
PROCEDURE TELL_IF_ONE (W : WORD);
{This procedure writes a message which}
{indicates whether W = 1}

  BEGIN
    IF W = 1 THEN
      BEGIN
        WRITELN ('W is indeed 1');
        RETURN
      END;
    WRITELN ('W is not 1');
  END;

FUNCTION FACT (I : INTEGER); WORD;
{This function returns the factorial of i}

  BEGIN
    FACT := 1;
    IF I < Ø THEN
      BEGIN
        WRITELN ('I MUST BE >= Ø. I is :', I);
        {the above is an error message}
        RETURN
      END;
      FOR VAR J = I DOWNTO 2 DO
        FACT := RESULT (FACT) * J;
  END;
```

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 can be a variable, constant identifier, structured constant, or function identifier; it cannot 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 can 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 can 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 detected 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 this version of 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.

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 this version of Pascal provide for such tests:

- AND THEN

  X AND THEN Y is false if X is false; Y is evaluated only if X is true.

- OR ELSE

  X OR ELSE Y is true if X is true; Y is evaluated only if X is false.

AND THEN and OR ELSE are logical operators similar to AND and OR, respectively.

If you use several sequential control operators, the compiler evaluates them strictly from left to right.

You can only include these operators in the Boolean expression of an IF, WHILE, or UNTIL clause; they cannot be used in other Boolean expressions. Furthermore, they cannot occur in parentheses and are evaluated after all other operators.

Examples of sequential control operators:

```
IF SYM <> NIL AND THEN SYM^.VAL < Ø THEN
  NEXT_SYMBOL;

WHILE I <= MAX AND THEN VECT [I] <> KEY DO
  I := I + 1;

REPEAT GEN (VAL)
UNTIL VAL = Ø OR ELSE (QU DIV VAL) = Ø;

WHILE POOR AND THEN GETTING POORER
  OR ELSE BROKE AND THEN BANKRUPT DO
    GET_RICH
```

This index covers both Volumes 1 and 2. Sections 1 through 12 are in Volume 1. Sections 13 through the Glossary are in Volume 2.

Page numbers in boldface indicate the principal discussion of a topic.

USER'S COMMENT SHEET

Pascal Reference Manual, Volume 1
Third Edition
A-09-00852-01-A

We welcome your comments and suggestions. They help us improve our manuals. Please give specific page and paragraph references whenever possible.

Does this manual provide the information you need? Is it at the right level? What other types of manuals are needed?

Is this manual written clearly? What is unclear?

Is the format of this manual convenient in arrangement, in size?

Is this manual accurate? What is inaccurate?

Name _____ Date _____

Title _____ Phone _____

Company Name/Department _____

Address _____

City _____ State _____ Zip Code _____

Thank you. All comments become the property of Convergent Technologies, Inc.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 1807 SAN JOSE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Convergent Technologies
Attn: Technical Publications
2700 North First Street
PO Box 6685
San Jose, CA 95150-6685

Fold Here

Please Do Not Staple

Tape

Convergent

2700 North First Street
San Jose, CA 95150–6685

Printed in USA