TOPS-20 PASCAL Language Manual

AA-L315A-TM

September 1983

This document describes the elements of the PASCAL language supported by TOPS-20 PASCAL.

OPERATING SYSTEM: TOPS-20 V5.1 (2040,2060)
TOPS-20 V4.1 (2020)

SOFTWARE: PASCAL V1.0
LINK V5.1
RMS V1.2

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center. Outside the United States, orders should be directed to the nearest DIGITAL Field Sales Office or representative.

Northeast/Mid-Atlantic Region	**Central Region**	**Western Region**
Digital Equipment Corporation	Digital Equipment Corporation	Digital Equipment Corporation
PO Box CS2008	Accessories and Supplies Center	Accessories and Supplies Center
Nashua, New Hampshire 03061	1050 East Remington Road	632 Caribbean Drive
Telephone:(603)884-6660	Schaumburg, Illinois 60195	Sunnyvale, California 94086
	Telephone:(312)640-5612	Telephone:(408)734-4915

digital equipment corporation • marlboro, massachusetts

First Printing, September 1983

CONTENTS

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual describes the PASCAL language and the PASCAL debugger, PASDDT, as they are implemented on the TOPS-20 operating system. This document is designed primarily for reference; it is not a tutorial document.

INTENDED AUDIENCE

This manual is intended for readers who know the PASCAL language. The reader need not have a detailed understanding of the TOPS-20 operating system, but some familiarity with either is helpful. For information about the TOPS-20 operating system, refer to the documents listed below under "Associated Documents." For introductory information about the use of the PASCAL language, refer to the _TOPS-20 PASCAL Primer_, Order Number AA-L314A-TM.

STRUCTURE OF THIS DOCUMENT

This manual contains the following chapters and appendixes:

- Chapter 1 provides an introduction to the use of PASCAL and describes the format of a PASCAL program.
- Chapter 2 introduces basic concepts including constants, variables, data types, expressions, and scope.
- Chapter 3 describes the components of an expression.
- Chapter 4 describes the program heading and declaration section.
- Chapter 5 describes the statements that perform the actions of the program.
- Chapter 6 explains the use of functions and procedures, and summarizes the predeclared functions and procedures supplied with the PASCAL-20 language.
- Chapter 7 provides detailed information on input and output procedures.
- Chapter 8 describes the compiling, loading, and executing of PASCAL programs on the TOPS-20 operating system.
- Chapter 9 describes the PASCAL-20 debugger, PASDDT.

- Appendix A lists the various messages you can receive.

- Appendix B lists the ASCII character set.

- Appendix C presents the PASCAL-20 language in the Backus-Naur form and includes syntax diagrams.

- Appendix D summarizes the extensions incorporated in the PASCAL-20 language.

- Appendix E describes how PASCAL-20 complies with the standard proposed by the International Standardization Organization (ISO).

- Appendix F summarizes the differences between TOPS-20 PASCAL and VAX-11 PASCAL.

- Appendix G describes the calling sequences and conventions used by PASCAL for user-defined procedures and functions.

ASSOCIATED DOCUMENTS

Associated manuals are:

- TOPS-20 PASCAL Primer

- TOPS-20 User's Guide

- TOPS-20 Commands Reference Manual

- LINK Reference Manual

- EDIT Reference Manual

- TV Editor Manual

CONVENTIONS USED IN THIS DOCUMENT

This document uses the following conventions:

Convention	Meaning
...	A horizontal ellipsis means that the preceding item can be repeated one or more times.
⋮	A vertical ellipsis means that not all of the statements in a figure or example are shown.
⟦ ⟧	Double brackets in statement and declaration format descriptions enclose optional items, for example: `WRITE ( ⟦OUTPUT,⟧ print-list)`
[ ]	Square brackets show where the syntax requires square brackets. This notation is used with arrays, for example: `ARRAY[1..5] OF INTEGER`
{ }	Braces enclose lists from which you must choose one item, for example: $\left\{\begin{matrix}\text{TO}\\\text{DOWNTO}\end{matrix}\right\}$
(RET)	This symbol indicates where you press the RETURN key.
<CTRL/x>	The notation <CTRL/x> indicates that you must press the key labeled CTRL while simultaneously pressing another key (x), for example, <CTRL/Z>.
UPPERCASE LETTERS	Uppercase letters in a command line indicate information that you must enter as shown.
lowercase letters	Lowercase letters in a command line indicate variable information you supply.
Simple_Procedure	In programming examples, all identifiers, (names created by the programmer), are printed in lowercase letters with initial capitals.
Contrasting Colors	Orange - where examples contain both user input and computer output, the characters you type are in orange; the characters printed on the terminal are in black.

CHAPTER 1

INTRODUCTION

PASCAL-20 is an extended implementation of level 1 of the standard proposed for the PASCAL language by the International Standardization Organization (ISO). This manual describes the use of PASCAL under the TOPS-20 operating system. PASCAL-20 includes all the language elements as defined in the PASCAL User Manual and Report by Jensen and Wirth, as well as the following extensions:

- Exponentiation operator
- Hexadecimal, octal, and binary integers
- Double-precision real data type
- Dollar sign ($) and underline (_) characters in identifiers
- External procedure and function declarations
- CARD, CLOCK, EXPO, SNGL, and UNDEFINED functions
- REM operator
- OTHERWISE clause in the CASE statement
- OPEN and CLOSE procedures for file access
- FIND procedure for direct access to sequential files
- Optional carriage control for output files
- DATE, TIME, HALT, and LINELIMIT procedures
- Variable initialization
- Separate compilation
- %INCLUDE directive for alternate input files during compilation
- Support for calling externally declared FORTRAN subroutines, and for declaration of PASCAL subroutines that can be called by FORTRAN

Refer to Appendix E for more information on ISO compliance.

This chapter presents an overview of the important concepts in PASCAL and illustrates the structure of a PASCAL program. It also describes PASCAL's lexical elements -- the character set, reserved words, identifiers, and special symbols. The final sections explain how to document your program and how to include existing files.

1.1 OVERVIEW OF PASCAL

A PASCAL program performs operations on data items known as constants, variables, and function designators. A constant is a quantity with an unchanging value. A constant to which you give a name is called a symbolic constant. A variable is a quantity whose value can change while the program executes. A function designator causes the execution of a group of statements that is associated with an identifier and returns a value. The function type is determined by the type of the value it returns.

1.1.1 Data Types

Every PASCAL data item is associated with a data type. A data type, which is usually indicated by a type identifier, determines both the range of values a data item can assume and the operations that can be performed upon it. In addition, the type implicitly indicates how much storage space is required for all possible values of the data item.

PASCAL provides identifiers for many predefined types. Thus, a program's operations can involve integers, real numbers, Boolean and character data, arrays, records, sets, and pointers to dynamic variables. PASCAL also allows you to create your own types by defining an identifier of your choice to represent a range of values.

The type of a constant is the type of its corresponding value. You establish variable and function types when you declare them. In general, they cannot change. Although variables and functions can change in value any number of times, all the values they assume must be within the range established by their type. A variable does not assume a value until the program assigns it one. A function is assigned a value during its execution.

PASCAL associates types not only with data items, but also with expressions. An expression is the computation of a value resulting from a combination of variables, constants, function designators, and operators. In PASCAL, you can form expressions using arithmetic, relational, logical, string, and set operators. Arithmetic expressions produce integer or real number values. Relational, logical, string, and most set expressions yield Boolean results. Other set expressions form the union, intersection, and differences of two sets.

1.1.2 Structure of a PASCAL Program

A PASCAL program consists of a heading and a block. The heading specifies the name of the program and the names of any external files the program may use. The block is divided into two parts: the declaration section, which contains data declarations; and the executable section, which contains executable statements. Figure 1-1 points out each part of a sample PASCAL program.

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

Declaration Section

```
         TYPE Yes_No = (Yes, No);
         VAR Subtotal, Operand : REAL;
             Equation : BOOLEAN;
             Operator : CHAR;
             Answer : Yes_No;

         PROCEDURE Instructions;
```

Procedure Block

```
                   BEGIN
                   WRITELN ('This program adds, subtracts, multiplies, and');
                   WRITELN ('divides real numbers. Enter a number in response');
                   WRITELN ('to the Operand: prompt and enter an operator -- ');
                   WRITELN ('+, -, *, /, or = -- in response to the Operator:');
                   WRITELN ('prompt. The program keeps a running subtotal');
                   WRITELN ('until you enter an equal sign (=) in response to');
                   WRITELN ('the Operator: prompt.  You can then exit from');
                   WRITELN ('the program or begin a new set of calculations.');
                   END; (*end of procedure Instructions*)
```

Executable Section

```
         BEGIN

         WRITE ('Do you need instructions? Type yes or no. ');
         READLN (Answer);
         If Answer = Yes THEN Instructions;

         REPEAT

                   Equation := FALSE;
                   Subtotal := 0;
                   WRITE ('Operand:');
                   READLN (Subtotal);

                   WHILE (NOT Equation) DO
                             BEGIN
                             WRITE ('Operator:');
                             READLN (Operator);
                             IF (Operator = '=') THEN
                             BEGIN
                             Equation := TRUE;
                             WRITELN ('The answer is ',Subtotal:5:2)
                             END
                             ELSE
                                   BEGIN
                                   WRITE ('Operand:');
                                   READLN (Operand);
                                      CASE Operator OF
                                         '+' : Subtotal := Subtotal + Operand;
                                         '-' : Subtotal := Subtotal - Operand;
                                         '*' : Subtotal := Subtotal * Operand;
                                         '/' : Subtotal := Subtotal / Operand
                                      END;
                                   WRITELN ('The subtotal is ',Subtotal:5:2)
                                   END
                             END;

                   WRITE ('Any more calculations? Type yes or no.');
                   READLN (Answer);
         UNTIL Answer = No;
         END.
```

MR-S-3150-83

Figure 1-1: Structure of a PASCAL Program

Procedure and function declarations have the same structure as programs. Note, in Figure 1-1, the heading and block of the procedure instructions. This manual uses the term subprogram to denote a procedure or function.

1.1.3 Definitions and Declarations

PASCAL requires you to define every constant and user-created type and to declare every label, variable, procedure, and function used in your program. The declaration section of the program contains LABEL, CONST, TYPE, VAR, VALUE, PROCEDURE, and FUNCTION sections, in which you define and declare all the data your program uses. All of these except LABEL introduce identifiers and indicate what they represent. LABEL declares numeric labels that correspond to executable statements accessed by the GOTO statement. PASCAL allows you to assign initial values to variables you declare in a VAR section. An initialized variable assumes the given value when program execution begins.

1.1.4 Executable Statements

The executable section of a PASCAL program contains the statements that specify the program's actions. The executable section is delimited by the reserved words BEGIN and END. Between BEGIN and END are conditional and repetitive statements, statements that assign values to variables and functions, and statements that control program execution.

1.1.5 Subprograms

PASCAL provides several ways for you to group together definitions, declarations, and executable statements. One way is to group them into procedures and functions, generically called subprograms. Both kinds of subprograms are groups of statements associated with an identifier. Procedures are usually written to perform a series of actions, while functions are written to compute a value.

Subprograms constitute a convenient way to isolate the individual tasks that the main program is to accomplish. Subprograms do not exist independently of the program; they are called either by an executable statement known as a procedure call or by a function designator appearing within an expression. PASCAL supplies many predeclared subprograms that perform commonly used operations, including input and output.

A subprogram consists of a heading and a block. The heading provides the name of the subprogram, usually a list of formal parameters that declare the input data for the program, and, in the case of functions, the type of the result. The subprogram block consists of an optional declaration section and an executable section. When the declaration section is present, it declares data that is local to the routine (that is, data that is unavailable outside the subprogram).

PASCAL is a block-structured language in that it allows you to nest subprogram blocks not only within the main program, but also within other subprograms. Each subprogram can make its own local definitions and declarations and can even redeclare an identifier that has been declared in an outer block. A subprogram declared at an inner level has access to the declarations and definitions made in all blocks that enclose it.

1.1.6 Compilation Units

A program is sometimes called a compilation unit in this manual because it can be compiled as a single unit (unlike a subprogram, which cannot be compiled without the context of a program). A program consists of a heading and a block, just as a subprogram does. The heading consists of the name of the program and possibly a list of identifiers to indicate any external files that the program uses. The declaration section of the program block declares data that is available at all program levels, including all nested subprograms.

1.2 LEXICAL ELEMENTS

A PASCAL program is composed entirely of lexical elements. These elements can be individual symbols, such as arithmetic operators; or they can be words that have special meaning to PASCAL. The basic unit of any lexical element is a character, which must be a member of the ASCII character set, as described in Section 1.2.1. Some characters are special symbols that PASCAL uses as statement delimiters, operators, and elements of the language syntax. Special symbols are listed in Section 1.2.4.

The words that PASCAL uses are combinations of alphabetic characters and occasionally a dollar sign ($), an underscore (_), or a percent sign (%). PASCAL reserves some words for the names of executable statements, operations, and some of the predefined data types. Reserved words are listed in Section 1.2.2. Other words in a PASCAL program are called identifiers. Predefined identifiers represent routines and data types provided by PASCAL. Other identifiers can be created by you to name programs, constants, variables, and any other necessary program segment that is not already named. Section 1.2.3 explains the use of both kinds of identifiers.

1.2.1 Character Set

PASCAL uses the full American Standard Code for Information Interchange (ASCII) character set (see Appendix B). The ASCII character set contains 128 characters in the following categories:

- The uppercase and lowercase letters A through Z and a through z
- The numbers 0 through 9
- Special characters, such as ampersand (&), question mark (?), and equal sign (=)
- Nonprinting characters, such as space, tab, line feed, carriage return, and bell

The PASCAL compiler does not distinguish between uppercase and lowercase characters, except in character and string constants and the values of character and string variables. For example, the reserved word PROGRAM has the same meaning when written as any of the following:

```
PROGRAM

PRogrAm

program
```

The constants below, however, represent different characters:

```
'b'

'B'
```

The following two constants represent different strings:

```
'BREAD AND ROSES'

'Bread and Roses'
```

1.2.2 Reserved Words

PASCAL reserves the words in Table 1-1 as names for statements, data types, and operators. This manual shows these words in uppercase characters.

Table 1-1: Reserved Words

AND	END	NIL	SET
ARRAY	FILE	NOT	THEN
BEGIN	FOR	OF	TO
CASE	FUNCTION	OR	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	PROCEDURE	VAR
DO	IN	PROGRAM	WHILE
DOWNTO	LABEL	RECORD	WITH
ELSE	MOD	REPEAT	

You can use reserved words in your program only in the contexts in which PASCAL defines them. You cannot redefine a reserved word for use as an identifier.

In PASCAL, the following words are considered semireserved words:

```
MODULE
OTHERWISE
REM
VALUE
```

Like the reserved words, PASCAL also predefines these semireserved words. However, unlike reserved words, you can redefine these words for your own purposes. If you redefine them, they can no longer be used for their original purpose within the scope of the block in which they are redefined.

1.2.3 Identifiers

PASCAL uses the term identifier to mean the name of a program, module, constant, type, variable, procedure, or function. An identifier is a sequence of characters that can include letters, digits, dollar signs ($), and underline symbols (_), with the following restrictions:

- An identifier can begin with any character other than a digit.
- An identifier must be unique in its first 31 characters within the block in which it is declared.
- An identifier must not contain any blanks.

PASCAL places no restrictions on the length of identifiers, but scans only the first 31 characters for uniqueness; the rest are ignored. The following are examples of valid and invalid identifiers:

Valid	Invalid
For2n8	4awhile (starts with a digit)
Max_Words	Up&to (contains the ampersand)
Upto	
$CREMBX	

1.2.3.1 Predeclared Identifiers - PASCAL predeclares some identifiers as names of functions, procedures, types, values, and files. These predeclared identifiers are listed in Table 1-2 and appear in uppercase characters throughout this manual.

Table 1-2: Predeclared Identifiers

ABS	EXP	OPEN	SINGLE
ARCTAN	EXPO	ORD	SNGL
BOOLEAN	FALSE	OUTPUT	SQR
CARD	FIND	PACK	SQRT
CHAR	GET	PAGE	SUCC
CHR	HALT	PRED	TEXT
CLOCK	INPUT	PUT	TIME
CLOSE	INTEGER	READ	TRUE
COS	LINELIMIT	READLN	TRUNC
DATE	LN	REAL	UNDEFINED
DISPOSE	MAXINT	RESET	UNPACK
DOUBLE	NEW	REWRITE	WRITE
EOF	ODD	ROUND	WRITELN
EOLN		SIN	

You can redefine a predeclared identifier to denote some other item. Doing so, however, means that you can no longer use the identifier for its usual purpose within the scope of the block in which it is redefined.

For example, the predeclared identifier READ denotes the READ procedure, which performs input operations. If you use the word READ to denote something else, such as a variable, you cannot use the READ procedure. Because you could lose access to useful language features, you should avoid redefining predeclared identifiers.

The directives FORTRAN, FORWARD, EXTERN, and EXTERNAL are also predeclared by the PASCAL compiler. However, they retain their meanings as directives even if you redefine them as identifiers.

1.2.3.2 **User Identifiers** - User identifiers denote the names of programs, modules, constants, variables, procedures, functions, and user-defined types. User identifiers name all significant data structures, values, and actions that are not represented by a reserved word, predeclared identifier, or special symbol.

1.2.4 Special Symbols

Special symbols represent arithmetic, relational, and set operators, delimiters, and other syntax elements. PASCAL includes the special symbols listed in Table 1-3.

Table 1-3: Special Symbols

Name	Symbol	Name	Symbol
Plus sign	+	Period	.
Equal	=	Multiplication	*
Not equal	<>	Less than	<
Exponentiation	**	Colon	:
Subrange operator	..	Comma	,
Parentneses	( )	Square brackets	[ ] (. .)
Comment	(* *) { }	Division	/
Minus sign	-	Greater than	>
Less than or equal	<=	Semicolon	;
Assignment operator	:=	Pointer	^ @
Greater than or equal	>=		

1.3 DELIMITERS

PASCAL uses two special symbols as delimiters: the semicolon (;) and the period (.). The semicolon separates one PASCAL statement from the next. One line of your program can contain one or many statements, but the statements must be separated by semicolons. The period marks the end of the PASCAL program.

The semicolon and the period are the only characters that PASCAL recognizes as delimiters. Spaces, tabs, and carriage-return/line-feed combinations are separators and cannot appear within an identifier, a number, or a special symbol. You must use at least one separator between consecutive identifiers, reserved words, and numbers; but you can use more than one if you want. You could, for instance, put each element of a PASCAL program on a separate line:

```
PROGRAM
Sim
(
OUTPUT)
;
BEGIN
WRITELN (
'This is a simple program.'
)
END.
```

You could also put the entire program on one line:

```
PROGRAM Sim(OUTPUT);BEGIN WRITELN('This is a simple program.')END.
```

As long as each complete statement is separated from the next by a semicolon, PASCAL interprets your input correctly. However, including spaces, tabs, and carriage-return/line-feed combinations make your program easier to read and understand. For readability, you could write it as follows:

```
PROGRAM Sim (OUTPUT);
  BEGIN
    WRITELN('This is a simple program.')
  END.
```

The reserved words BEGIN and END are also used as delimiters. BEGIN indicates the start of the executable section or a compound statement, and need not be followed by a semicolon.

END indicates the end of one of the following:

- A record definition
- An executable section
- A compound statement
- A CASE statement (see Section 5.3.3)

Although PASCAL does not require one, you can use a semicolon immediately before END. A semicolon in this position results in an empty statement between the semicolon and the reserved word END. The empty statement implies no action.

1.4 DOCUMENTING YOUR PROGRAM

In addition to statements and delimiters, you can put comments in your PASCAL program. Comments are simply words or phrases that describe what happens in the program.

You can enclose comments in braces { }, as follows:

```
{ This is a comment. }
```

Also, you can start a comment with the left-parenthesis/asterisk character pair, and end it with the asterisk/right-parenthesis character pair, as follows:

```
(* This is also a comment *)
```

You can also mix the type of comment characters you use. For example, you can use a left brace with an asterisk/right-parenthesis character pair:

```
{ This is another comment *)
```

You can place a comment anywhere a space is legal. Unlike statements, comments are not delimited by semicolons.

A comment can contain any ASCII character because PASCAL ignores the text of the comment.

NOTE

> To turn off braces { } as recognized comment characters, use the /NATIONAL switch. See Section 8.4.3 for more information on this switch.

1.5 THE %INCLUDE DIRECTIVE

The %INCLUDE directive allows you to access statements from a PASCAL file, called the included file, during compilation of the current file. The contents of the included file are inserted in the place where the PASCAL compiler finds the directive. This directive can appear anywhere in the PASCAL program.

Format

$$\text{%INCLUDE 'file specification } \left[\left\{ \begin{array}{l} \text{/LIST} \\ \text{/NOLIST} \end{array} \right\}\right] \text{'}$$

where:

'file specification'	is the name of the file to be included. The apostrophes are required.
/LIST	indicates that the included file is to be printed in the listing. This is the default.
/NOLIST	indicates the included file is not to be printed in the listing.

When the compiler finds the %INCLUDE directive, it stops reading from the current file and begins reading from the included file. When the compiler reaches the end of the included file, it resumes compilation immediately following the %INCLUDE directive.

This directive can appear wherever a comment can appear. An included file can contain any PASCAL declarations or statements. However, the declarations in an included file, when combined with the other declarations in the compilation, must follow the required order for declarations.

In the following example, the %INCLUDE directive specifies the file CONDEF.PAS, which contains constant declarations.

Main PASCAL Program

```
PROGRAM Student_Courses (INPUT, OUTPUT, SCHED);

CONST %INCLUDE 'CONDEF.PAS'

TYPE Schedules = RECORD
                 Year : (Fr, So, Jr, Sr);
                 Name : PACKED ARRAY [1..30] OF CHAR;
                 Parents : PACKED ARRAY [1..40] OF CHAR;
                 College : (Engineering, Architecture, Agriculture)
                 END;
   .
   .
   .
```

External File
CONDEF.PAS

```
MAX_CLASS = 300;
N_PROFS = 140;
FROSH = 3000;
```

The %INCLUDE directive instructs the compiler to insert the contents of the file CONDEF.PAS after the reserved word CONST in the main program. The main program Student_Courses is compiled as if it contained the following:

```
     PROGRAM Student_Courses (INPUT, OUTPUT, SCHED);

       CONST Max_Class = 300;
             N_Profs = 140;
             Frosh = 3000;

       TYPE Schedules = RECORD
                        Year: (Fr, So, Jr, Sr);
                        Name: PACKED ARRAY [1..30] OF CHAR;
                        Parents : PACKED ARRAY [1..40] OF CHAR;
                        College : (Engineering, Architecture, Agriculture)
                        END;
       .
       .
       .
```

You can use the %INCLUDE directive in another included file; however, recursive %INCLUDE directives are not allowed. If, for example, the file OUT.PAS contains an %INCLUDE directive for the file IN.PAS, then IN.PAS must not contain an %INCLUDE directive for the file OUT.PAS.

An included file at the outermost level of a program is said to be included at the first level. A file included by a first-level included file is at the second level, and so on. There is no limit to the number of included files you can nest in a program. Figure 1-2 illustrates some levels of included files.

```
Main Program                     A.PAS
------------                     -----

Program P;                       CONST %INCLUDE 'B.PAS'

TYPE %INCLUDE 'A.PAS'            VAR %INCLUDE 'C.PAS'
(* Level 1 *)                    (* Both Level 2 *)

C.PAS                            D.PAS
-----                            -----

VAR %INCLUDE 'D.PAS'             FUNCTION %INCLUDE 'E.PAS'
(* Level 3*)
                                 PROCEDURE %INCLUDE 'F.PAS'
                                 (* Both Level 4 *)

F.PAS                            G.PAS
-----                            -----

FUNCTION %INCLUDE 'G.PAS'
```

Figure 1-2: %INCLUDE File Levels

CHAPTER 2

PASCAL DATA TYPES

This chapter describes PASCAL data types and how to define and declare them in the TYPE and VAR sections of a PASCAL program. This chapter also provides general information about using each data type.

PASCAL uses three categories of data types:

1. Scalar

2. Structured

3. Pointer

Scalar data types represent ordered groups of values. The scalar data types, which are described in Section 2.2, consist of predefined and user-defined data types. Predefined data types include integers, real numbers, and characters. User-defined data types include a range of explicitly defined values and a subrange of another data type. Scalar data types are building blocks for the structured data types.

Structured data types are collections of data types organized in specific ways. Structured data types include arrays, record files, and sets. These are described in Section 2.3.

Pointer data types provide access to dynamic data structures. They are described in Section 2.4.

2.1 DECLARING DATA TYPES

PASCAL provides two methods of declaring variables of a particular type. You can define the type in the TYPE section, and then use a declaration in the VAR section to declare one or more variables of the newly defined type. The general format is:

```
TYPE type identifier = type definition;
VAR variable name : type identifier;
```

Alternatively, you can declare a variable by specifying the type definition in the VAR section and omitting the type identifier and type definition from the TYPE section. The general format for this method is the following:

```
VAR variable name :  type definition;
```

If a data type is used only once within the program, it is simpler to define it in the VAR section.

If a data type is used more than once in the program, it is more efficient to define the data type within the TYPE section. This creates a structure that can be accessed by more than one identifier. For example, if a program uses an array three times, you can define the array type in the TYPE section, and assign three identifiers to that array type in the VAR section.

2.2 SCALAR TYPES

Scalar data types consist of ordered sets of values with the concept of predecessor and successor. For example, the scalar data type INTEGER represents whole numbers that follow in a predefined sequence: 5 is less than 300. Scalar data types encompass two subclasses: predefined and user-defined. These are described in the following sections.

2.2.1 Predefined Data Types

PASCAL provides the following predefined scalar data types:

1. INTEGER
2. REAL
3. SINGLE
4. DOUBLE
5. BOOLEAN
6. CHAR

The predefined types SINGLE and DOUBLE provide explicit single-precision and double-precision real numbers. Throughout this manual, the term REAL refers to REAL, SINGLE, and DOUBLE types.

The following sections describe each predefined data type.

2.2.1.1 INTEGER Data Type - The type INTEGER denotes positive and negative whole number values ranging from (-2**35) to (+2**35)-1, or -34359738368 to +34359738367. The largest possible value of the INTEGER data type is known by the predefined constant identifier MAXINT.

You can indicate a decimal integer constant with decimal digits combined with plus and minus signs. The following are valid decimal constants in PASCAL:

```
   17
 -333
    0
   +1
89324
```

A minus sign (-) must precede a negative integer value. A plus sign (+) may precede a positive integer, but the sign is not required. No commas or decimal points are allowed.

In addition to decimal notation, PASCAL allows you to specify integer constants in binary, octal, and hexadecimal notation. You can use constants written in these notations anywhere that decimal integer constants are permitted.

To specify an integer constant in binary, octal, or hexadecimal notation, place a percent sign (%) and a letter in front of a number enclosed in apostrophes. The appropriate letters, which can be either uppercase or lowercase, are B for binary notation, O for octal notation, and X for hexadecimal notation. An optional plus or minus sign can precede the percent sign to indicate a positive or negative value. Note that regardless of which notation you use, the value can not exceed MAXINT, for example:

```
-%B'111001'
%b'10000011'
%O'7712'
-%o'473'
+%X'53A1'
-%x'DEC'
```

2.2.1.2 REAL Data Type - The reserved words REAL, SINGLE, and DOUBLE denote the real number types. In PASCAL, a real number can range from +-0.14*10**-38 through +-3.4*10**38, with a typical precision of eight decimal digits. REAL and SINGLE are synonymous; both have single-precision real number values. The type DOUBLE allows you to declare double-precision real variables. You can assign real and integer values to a variable of type REAL, SINGLE, or DOUBLE. If you assign an integer value to a variable of type REAL, PASCAL converts the integer to a real number.

In a PASCAL program, you can write real numbers in two ways; fixed on floating point. With fixed point notation, you write the number with the decimal point exactly where it appears in the value. The first way is the following form:

```
  2.4
893.2497
 -0.01
  8.0
-23.18
  0.0
```

Note that, in this form, at least one digit must appear on each side of the decimal point. That is, a zero must always precede the decimal point of a number between 1 and -1, and a zero must follow the decimal point of a whole number.

Some numbers, however, are too large or too small to write conveniently in the above format. PASCAL provides scientific (also known as exponential) notation as a second way of writing real numbers. In scientific notation, you write the number as a positive or negative value followed by an exponent, for example:

```
   2.3E2
  -0.07E4
  10.0E-1
-201E+3
    -2.14159E0
```

The letter E after the value means that the value is to be multiplied by a power of 10. Note that you can use an uppercase or lowercase letter. The integer following the E gives the power of 10; the integer can be positive or negative. Using scientific notation, you can write the real number 237.0 in any of the following ways:

```
237e0
2.37E2
0.000237E+6
2370E-1
0.0000000237E10
```

This format is often called floating-point format because the implied position of the decimal point "floats" depending on the exponent following the E. At least one digit must appear on each side of the decimal point, if the decimal point is present.

PASCAL provides single and double-precision representation for real numbers. Single precision typically provides eight significant digits, depending on the magnitude of the number. Double precision extends the number of significant digits to 18.

To indicate a double-precision value, you must use floating-point notation, replacing the letter E with an uppercase or lowercase D, for example:

```
 0D0
 4.371528665D-3
-812d2
 4D-3
```

The integer following the D is an exponent, as in single-precision floating-point numbers. All the above values have approximately 18 significant digits.

2.2.1.3 BOOLEAN Data Type

- BOOLEAN data types can have the value TRUE or FALSE. Boolean values are the result of testing expressions for truth or validity. The result of a relational expression (for example, A < B) is a Boolean value.

PASCAL defines Boolean data types as predefined identifiers and orders them so that FALSE is less than TRUE. For assignment purposes, the type BOOLEAN is compatible with those variables and expressions that yield a BOOLEAN result.

2.2.1.4 CHAR Data Type

- The value of data type CHAR is a single value from the ASCII character set, as listed in Appendix B. To specify a character value, enclose an ASCII character in apostrophes. The apostrophe character itself must be typed twice within apostrophes. Each of the following is a valid character value:

```
'A'
'z'
'0'
'.'
''''
'?'
```

You can use strings such as 'HELLO' and '****', but you must represent them as packed arrays of characters (see Section 2.3.1.2). When you use the ORD function in an expression of type CHAR, the result is the ordinal value in the ASCII character set of the character value. See Section 2.2.2.3 for an explanation of the ORD function.

2.2.2 User-Defined Scalar Data Types

User-defined scalar data types are those that you define, as opposed to those data types that PASCAL predefines for you. PASCAL allows you to define two kinds of scalar data types: enumerated and subrange. An enumerated type consists of an ordered list of identifiers. The subrange type is a continuous range of values of a defined scalar type, called a base type. The following sections describe these two user-defined types.

2.2.2.1 Enumerated Data Types - An enumerated data type is an ordered list of identifiers. To define an enumerated type, list in some order all the identifiers denoting its values. With PASCAL, you can define an enumerated data type in two ways:

Format 1

```
    TYPE identifier = (identifier  [[,identifier, ...]] )
```

Format 2

```
    VAR identifier :  (identifier  [[,identifier, ...]] )
```

where:

identifier	is the name of the enumerated type.

For example:

```
    TYPE Beverage = (Milk, Water, Cola, Beer);
```

This TYPE section defines the type Beverage and lists all the values that Beverage can assume within a program.

PASCAL assigns an order to the items in your list from left to right. Thus, the values of an enumerated type follow a left-to-right order, so that the last value in the list is greater than the first, for example:

```
    TYPE Seasons = (Spring, Summer, Fall, Winter) ;
```

The relational expression (Spring < Fall) is TRUE because Spring precedes Fall in the list of values.

The only restriction on the values of an enumerated type is that you cannot define the same value in more than one list in the same TYPE section. For example, the following is illegal:

```
    TYPE Seasons = (Spring, Summer, Fall, Winter) ;
         Schoolyear = (Fall, Winter, Spring);
```

To initialize a variable of an enumerated type, specify a constant value. For example, you can assign the variable Quarter of type Seasons as follows:

```
     VAR Quarter :  Seasons := Fall;
```

The variable Quarter takes on the initial value Fall.

Examples

```
     TYPE Colors = (Red, Yellow, Green, Purple, Blue);
          Sport = (Swim, Run, Ski);
          Beverage = (Milk, Water, Cola, Beer);

     VAR Cookie :  (Oatmeal, Choc-Chip, Peanut-Butter, Sugar) := Sugar;
          Exercise, Fun :  Sport := Ski;
          Drink :  Beverage;
```

The TYPE section defines the types Colors and Sport, listing all the values that variables of each type can assume.

The VAR section declares the variable Cookie, which can have the values Oatmeal, Choc-Chip, Peanut-Butter, and Sugar. The variables Exercise and Fun are declared as type Sport, and Drink is declared as type Beverage.

Initial values are established for the identifiers Cookie, Exercise, and Fun in the VAR section.

2.2.2.2 Subrange Data Types

2.2.2.2 Subrange Data Types - A subrange specifies a limited portion of another scalar type (called the base type) for use as a type.

Format 1

```
     TYPE identifier = lower limit..upper limit
```

Format 2

```
     VAR identifier :  lower limit..upper limit
```

where:

identifier	is the name of the subrange.
lower limit	is the constant value at the low end of the subrange.
..	separates the limits of the subrange.
upper limit	is the constant value at the high end of the subrange.

The subrange type is defined only for the values between and including the lower and upper limits. The limits you specify must be constants; they cannot be expressions. (See Chapter 3 for information on expressions.) The values in the subrange are in the same order as in the base type.

The base type can be any enumerated or predefined scalar type except REAL. You can use a subrange type anywhere in the program where its base type is legal. The rules for operations on a subrange are the same as the rules for operations on its base type. A subrange and its base type are compatible.

The use of subrange types can make a program clearer. For example, integer values for the days of the year range from 1 to 365. Any value outside this range is obviously incorrect. You could specify an integer subrange for the days of the year as follows:

```
VAR Day-Of-Year :  1..366
```

By specifying a subrange, you indicate that the values of the variable Day-Of-Year are restricted to the integers from 1 to 366.

Example

```
TYPE Months = (Jan, Feb, Mar, Apr, May, Jun,
               Jul, Aug, Sep, Oct, Nov, Dec);

VAR Camp_Mos :  May..Oct;
Leaf_Mos :  Sep..Nov;
First_Half :  'A'..'M';
Word :  0..65535;
```

This example defines the variables Camp_Mos and Leaf_Mos as subranges of the enumerated type Months. A Camp_Mos value can be only May, Jun, Jul, Aug, Sep, or Oct. A Leaf_Mos value can be only Sep, Oct, or Nov. The variable First_Half is a subrange of the ASCII characters, with possible values uppercase A through uppercase M. The variable Word is a subrange of the integers from 0 to 65535.

2.2.2.3 The ORD Function

Each element of a scalar type (except the REAL type) has a unique ordinal value, which indicates its order in a list of elements of its type. The ORD function returns the ordinal value as an integer, for example:

```
ORD('Q')
```

This expression returns 81, which is the ordinal value of uppercase Q in the ASCII character set (see Appendix B). Note that the order of the ASCII character set may not be what you expect. The numeric characters are in numeric order, and the alphabetic characters are in alphabetic order. All uppercase characters have lower ordinal values than all lowercase characters, for example:

```
ORD('Q') is less than ORD('q') and
ORD('A') is less than ORD('Z') but
ORD('Z') is less than ORD('a')
```

You can use ORD on a value of an enumerated type. Enumerated types are ordered starting at zero, for example:

```
ORD(Tuesday)
```

Assuming that Tuesday is a value of type Weekdays (which includes the values Monday, Tuesday, Wednesday, Thursday, and Friday), this expression returns the integer 1.

The ordinal value of an integer is the integer itself. For example, ORD(0) equals 0, ORD(23) equals 23, and ORD(-1984) equals -1984.

2.3 STRUCTURED DATA TYPES

A structured data type consists of a collection of related data components; it is characterized by its method of structuring and its components. All structured data types consist of a collection of elements or components that are grouped together in a structure in which they can collectively be manipulated.

PASCAL provides four structured data types:

- ARRAY
- RECORD
- SET
- FILE

An array is a group of components of a predefined size and of the same type. A record consists of one or more named fields, each of which contains one or more data items. Records can include fields of different data types. A set is a collection of data items of the same scalar type, the base type. You can access a set as an entity, but you cannot access the set components as individual components or variables. A file is a sequence of data components that are of the same type; each component can be individually accessed. A file can be of variable length.

Section 2.3.1 describes arrays; Section 2.3.2 describes records; Section 2.3.3 describes sets; and Section 2.3.4 describes files.

2.3.1 Array Types

An array is a group of components of the same type that share a common name. You refer to each component of the array by the array name and an index (or subscript). An array type definition specifies the type of the indices and the type of the components.

Format

```
     ARRAY [index type  ⟦,index type...⟧  ] OF component type
```

where:

index type	specifies the type of the index. The index type can be a subrange, CHAR, BOOLEAN, or enumerated type; but it cannot be a REAL type.
component type	specifies the type of the components of the array.

The components of an array can be of any type. For example, you can define an array of integers, an array of records, or an array of real numbers. You can also define an array of arrays, which is known as a multidimensional array.

The indices of an array must be of a scalar type, but cannot be real numbers. Note that you cannot specify the type INTEGER as the index type. To use integer values as indices, you must specify an integer subrange, unless you are using conformant-array parameters. If necessary, PASCAL determines the subrange. For example, if the index is BOOLEAN, then PASCAL converts the subrange to FALSE..TRUE because the type BOOLEAN has only two legal values. For more information about conformant-array parameters, refer to Section 6.3.2.

The range of the index type establishes the size of the array and the way it is indexed, for example:

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

    VAR Let1 : Letters;
```

The array variable LET1 has 10 components, referred to as LET1[1], LET1[2], LET1[3], and so on, through LET1[10].

You can use array components in expressions anywhere you can use variables of the component type. For the array as a whole, however, you can use only the assignment statement (:=). An exception to this rule is made for character strings, which PASCAL defines as packed arrays of type CHAR. See Section 2.3.1.2.

2.3.1.1 Multidimensional Arrays

- An array with components of an array type is a multidimensional array. An array can have any number of dimensions, and each dimension can have a different index type. For example, the following declares a two-dimensional array variable:

```
    VAR Two_D : ARRAY [0..4] OF ARRAY ['A'..'D'] OF INTEGER;
```

PASCAL allows you to abbreviate the definition by specifying all the index types in one pair of brackets, for example:

```
    VAR Two_D : ARRAY [0..4,'A'..'D'] OF INTEGER;
```

To refer to a component of this array, you specify two indices, one integer and one character, in the order they were declared: Two_D[0,'A'], Two_D[0,'B'], and so on. You can also specify Two_D[0]['A']. The first index indicates the rows of the array, and the second index indicates the columns. Hence, you can picture the array Two_D as in Figure 2-1.

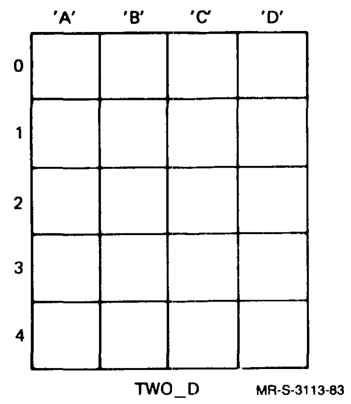


Figure 2-1: Two-Dimensional Array Two_D

When referring to the components of Two_D, the first component in the first row is Two_D[0,'A']. The second component in this row is Two_D[0,'B']. The first component in the second row is Two_D[1,'A']. The last component in the last row is Two_D[4,'D']. In general, element j of row i is Two_D[i,j].

You can define arrays of three or more dimensions in a similar fashion, for example:

```
    TYPE Chessmen = (QR,QN,QB,Q,K,KB,KN,KR,P,E);(*E means empty square*)

    VAR Chess3D : ARRAY [1..3, 1..8, QR..KR] OF Chessmen;
```

This declaration specifies a three-dimensional chess game. The indices of the array are the levels, the ranks, and the files of the chessboard. For example, the reference Chess3D [1, 1, QR] specifies the first level, first square in the upper left corner (bottom level, first rank, Queen's Rook file). Figure 2-2 illustrates the three levels of this array.

Chess3D(1,n, Chessmen) (bottom)	Chess3D(2,n, Chessmen) (middle)	Chess3D(3,n, Chessmen) (top)

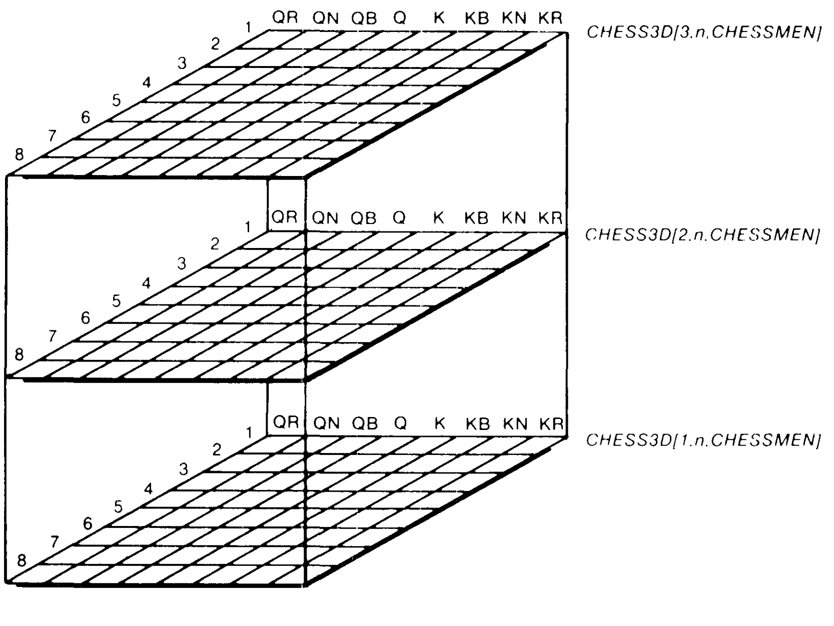


Figure 2-2: Three-Dimensional Array Chess3D

When storing values in an array, PASCAL increments the indices from right to left. Thus, PASCAL increments the rightmost index until the maximum value is reached, then moves to the left to the next index, and so on, until all indices have been incremented to the specified amount.

In the three-dimensional array Chess3D, PASCAL starts by holding the first two indices constant while stepping through the values of Chessmen. Thus, the first values are assigned to components Chess3D [1,1,QR] through Chess3D [1,1,KR]. Next, the second index is incremented and values are assigned to the components Chess3D [1,2,QR] through Chess3D[1,2,KR]. After these eight elements are assigned, the second index is again incremented, and values are assigned to Chess3D [1,3,QR] through Chess3D [1,3,KR]. The assignment process continues with the first index held constant until the second index has been incremented from 1 to 8. Then, the first index is incremented, and the process is repeated. Hence, all values for the bottom level (denoted by Chess3D[1,n,Chessmen]) are stored before any values for the middle level (denoted by Chess3D [2,n,Chessmen]). The top level (denoted by Chess3D[3,n,Chessmen]) receives its values last. Figure 2-3 illustrates this order.

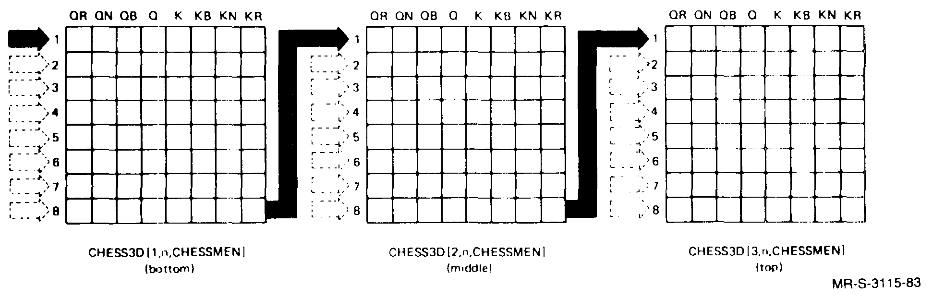


Figure 2-3: Storing Components in an Array

2.3.1.2 String Variables - A character string variable in PASCAL is defined as a packed array of characters with a lower bound of 1. To declare a string variable, specify a packed array of the proper length, for example:

```
    VAR NAME : PACKED ARRAY [1..20] OF CHAR;
```

This declaration allows you to store a string of 20 characters in the array variable NAME. The length of this string must be exactly 20 characters. PASCAL neither adds blanks to extend a shorter string nor truncates a longer string.

You can assign to a string variable the value of any string constant or variable of the defined length. You can also compare strings of the same length with the relational operators <, <=, >, >=, =, and <>. The result of a string comparison depends on the ordinal value (in the ASCII character set) of the corresponding characters in the strings, for example:

```
    'motherhood' > 'apple pies'
```

This relational expression is TRUE because lowercase 'm' comes after lowercase 'a' in the ASCII character set. If the first characters in the strings are the same, PASCAL looks for differing characters, as in the following:

```
    'string1' < 'string2'
```

This expression is also TRUE because the digit 1 precedes the digit 2 in the ASCII character set.

To assign a string constant to an array in the executable section, use an assignment statement. The string variable must be of the same size as the array; otherwise, an error occurs. The following example shows an assignment statement in the executable section:

```
TYPE String = PACKED ARRAY[1..10] OF CHAR;
VAR  Word : String;

BEGIN
   Word := 'orange    ';
   .
   .
   .
END;
```

The string constant 'orange ' is padded with spaces to match the size of the array.

The READ and READLN statements automatically pad the string constant if necessary. Thus, it is not necessary to pad the string constant with spaces to match the variable size, when using the predefined file INPUT or reading from a file defined as TEXT.

2.3.1.3 Initializing and Assigning Values to Arrays - You should assign values to an array either in the declarations or the executable section before using the array within the program. As with all variables, the value for each component is undetermined, until a value is specifically assigned.

To assign values to an array in the executable section, a value must be assigned to each component in the array. One method of doing this is to use a FOR statement. By using the FOR statement control variable as the array index, it is possible to step through all components of the array, setting them to the same initial value. Each index for the array is incremented at the same time the counter in the FOR statement is incremented. An example of this is:

```
VAR Array_Example : ARRAY [1..10] OF INTEGER;
    Index : INTEGER;

BEGIN
  FOR Index := 1 TO 10 DO
       Array_Example [Index] := 0;
  .
  .
  .
END;
```

In this example, the array has been defined as being an array of type INTEGER. As the FOR loop executes each time, the counter is incremented by one. Likewise, the index is incremented by one. On each execution of the loop, the current component is assigned a value of 0.

An array can be initialized in the VAR section. The following example shows each component of the array Array_Example being initialized with the value 0:

```
VAR Array_Example :  ARRAY [1..10] OF INTEGER := (10 of 0);
```

To assign values to a two-dimensional array in the executable section, you can use two nested FOR statements to increment the two indices, as shown in the following example:

```
     CONST Zero = 0;
     VAR Table : ARRAY [1..10,1..5] OF INTEGER;
         index_1,index_2 : INTEGER;

     BEGIN
          FOR index_1 := 1 TO 10 DO
              FOR index_2 := 1 TO 5 DO
                  Table[index_1,index_2] := Zero;
          .
          .
          .
     END;
```

Array_Example is defined to be a two-dimensional array of type INTEGER.

The nested FOR statements assign the value of Zero (which has been assigned the value of "0" in the CONST section) to each component in the array.

To initialize a two-dimensional array in the VAR section, specify a constructor for each row, in parentheses. The following example shows a two-dimensional array that is initialized in the VAR section:

```
VAR Table :  ARRAY [1..10,1..5] OF INTEGER := (10 OF (5 OF 0));
```

To assign values to arrays of three or more dimensions in the executable section, use three or more nested FOR statements:

```
     CONST Zero = 0;
     VAR  Table : ARRAY [1..5,1..3,1..2] OF INTEGER;
          index_1,index_2,index_3 : INTEGER;

     BEGIN
          FOR index_1 := 1 TO 5 DO
              FOR index_2 := 1 TO 3 DO
                  FOR index_3 := 1 TO 2 DO
                      Table[index_1,index_2] := Zero;
          .
          .
          .
     END;
```

This example shows the initialization of a three-dimensional array. The value of Zero is assigned to each element in Table, from Table[1,1,1] to Table[5,3,2].

To initialize an array of three or more dimensions in the VAR section, specify a constructor for each row. The following example shows a three-dimensional array being initialized in the VAR section:

```
VAR Table : ARRAY [1..5,1..3,1..2] OF INTEGER :=
                                    (5 OF (3 OF (2 OF ' ')));
```

2.3.1.4 Array Type Compatibility - You can assign one array to another only if the arrays are either identical or compatible. Arrays of the same type or equivalent types are identical. The following example demonstrates identical arrays:

```
TYPE Salary = ARRAY [1..50] OF REAL;
       Pay = Salary;

VAR Wage, Income : Salary;
      Money : Pay;
```

The arrays Wage and Income are identical because both are of type SALARY. The array Money of type PAY is identical to Wage and Income because the type PAY is declared equivalent to the type SALARY. Identical arrays are always compatible.

Arrays that are not identical are compatible if they meet all of the following criteria:

- They have the same number of components.
- Their elements are of compatible types.
- Their indices are of compatible types.
- The upper bounds of their indices are equal.
- The lower bounds of their indices are equal.
- Both are packed or neither is packed.
- For packed arrays of subrange types, the bounds of the subranges must be the same for both types.

The following two array types, though not identical, are compatible:

```
TYPE Grades = ARRAY [1..28] OF 0..4;
Feb_Temps = ARRAY [1..28] OF INTEGER;
```

Both types define arrays with 28 components, indexed from 1 to 28. The integer subrange components of type GRADES are compatible with the integer elements of type Feb_Temps. Therefore, you can assign variables of type GRADES to variables of type Feb_Temps, and vice versa. Note that, if the TYPE definition specified packed arrays, the types GRADES and Feb_Temps would not be compatible.

PASCAL does not check for valid assignments to subranges that are part of a structured type. If you assign an array of type Feb_Temps to one of type GRADES, you must ensure that the values are in the correct range. An out-of-range assignment does not result in an error message, even if the CHECK option is enabled at compile time.

2.3.1.5 Array Examples -

Example 1

```
     TYPE Times = 1..10;
     VAR Raceresults : ARRAY[1..50] OF Times;
          I : INTEGER;

     BEGIN
       FOR I := 1 TO 50 DO
            Raceresults[I] := 0
     END;
```

This example declares the variable Raceresults as a 50-component array of Times. The FOR statement assigns zero to each component in the array.

Example 2

```
     TYPE String = PACKED ARRAY [1..10] OF CHAR;
     VAR Composer, Word, Empty : String;

     BEGIN
         Word := 'engrossing';
         Composer := 'C.P.E.Bach';
         Empty := '
     END;
```

This example declares three string variables. It assigns string constants to the variables Word and Composer, and assigns a string of 10 spaces to the variable Empty.

Example 3

```
     CONST Days = 31;
     TYPE  Weather = (Rain, Snow, Sunny, Cloudy, Foggy);
           Month = ARRAY [1..DAYS] OF Weather;
```

This example shows how you can use a constant identifier in the index type. The indices of arrays of type Month range from 1 to the value of the constant Days.

2.3.2 Record Types

The record is a convenient way to organize several related data items of different types. A record consists of one or more fields, each of which contains one or more data items. Unlike the components of an array, the fields of a record can be of different types. The record type definition specifies the name and type of each field.

Format

```
RECORD

{ field id : type  ⟦ ;field id : type...⟧ ; ⟦variant clause ⟧  }
{ variant clause }

END;
```

where:

field id	specifies the names of one or more fields. The names must be identibfiers and must be separated by commas.
type	specifies the type of the corresponding field(s). A field can be any type.
variant clause	specifies the variant part of the record. See Section 2.3.2.1 for the format of a variant clause.

The names of the fields must be unique within the record, but can be repeated in different record types. For instance, you could define the field NAME only once within a particular record type. Other record types, however, could also have fields called NAME.

The values for the fields are stored in the order in which the fields are defined, for example:

```
VAR Team_Rec : RECORD
               Wins : INTEGER;
               Losses : INTEGER;
               Percent : REAL
               END;
```

The values for these fields are stored in the order Wins, Losses, Percent.

To refer to a field within a record, specify the name of the record variable and the name of the field, separated by a period. For example, Team_Rec.Wins, Team_Rec.Losses, and Team_Rec.Percent refer to the three fields of the record Team_Rec declared above. You can specify a field anywhere in the program that a variable of the field type is allowed. Thus, you could write:

```
Team_Rec.Wins := 9;

Team_Rec.Losses := 4;
```

Records can include fields that are themselves records, for example:

```
VAR Order : RECORD
            Part : INTEGER;
            Received : RECORD
                       Month : (Jan, Feb, Mar, Apr, May, Jun,
                                Jul, Aug, Sep, Oct, Nov, Dec);
                       Day : 1..31;
                       Year : INTEGER
                       END;
            Inventory : INTEGER
            END;
```

The fields in this record are referred to as Order.Part, Order.Received.Month, Order.Received.Day, Order.Received.Year, and Order.Inventory. The WITH statement provides an abbreviated notation for specifying the fields of a record (see Section 5.5).

2.3.2.1 Records with Variants - To allow a record to contain different data types at different times, you can define a record variant. To do this, specify one or more variants in the TYPE definition. A variant is a field or group of fields that can contain a different type or amount of data at different times during execution. Thus, two variables of the same record type can contain different types of data.

To specify a variant, include a variant clause in the record type definition. The variant clause must be the last field in the record.

Format

```
     CASE tag field OF
        case-label list : ( ⟦field id : type⟧  ⟦;field id : type...⟧ );
              .
              .
              .
```

where:

tag field	indicates the current variant of the record. You can specify the tag field in two ways: 1. tag name : tag type If you use this form, the tag field is a field in the record that is common to all variants. Tag name and tag type define the name and type of this field. The tag type can be any scalar type except a REAL type. You can use the tag field in the same way that you use any other field in the record; that is, you can use the record.fieldname format. 2. tag type If you use this form, you must keep track of the currently valid variant. The tag type can be any scalar type except a REAL type.
case-label list	specifies one or more constants of the tag field type.
field id	specifies the names of one or more fields. The field names must be identifiers and must be separated by commas. Note that, instead of the field identifiers, you can specify another variant clause, as in the last example in this section.
type	specifies the type of the variant field. The type cannot be a FILE type.

When you specify the tag field in the first form (tag name : tag type), you should reference only the fields in the currently valid variant. The following example shows the use of this form:

```
TYPE Name = PACKED ARRAY [1..20] OF CHAR;
     Day = (Mon, Tue, Wed, Thu, Fri);
     Stock = RECORD
             Part : 1..9999;
             Stock_Quantity : INTEGER;
             Supplier : Name;
             CASE Onorder : BOOLEAN OF
                             TRUE :(Promised : Day;
                                    Order_Quantity : INTEGER;
                                    Price: REAL);
                             FALSE :(Last_Shipment : Day;
                                    Rec_Quantity : INTEGER;
                                    Cost : REAL)
             END;
```

In this example, the last three fields in the record type vary depending on whether the part is on order. The tag name Onorder is defined in the variant clause. Records for which the value of Onorder is TRUE will contain information about the current order. Records for which this variable is FALSE will contain information about the previous shipment.

In the second way of specifying the tag field, you use only a tag type, as in this example:

```
TYPE Name = PACKED ARRAY [1..20] OF CHAR;
     Date = INTEGER;
     Sex = (Female, Male);
     Hosp = RECORD
            Patient : Name;
            Birthdate : Date;
            Age : INTEGER;
            CASE Sex OF
                 Female : (Births : 1..30);
                 Male : ()
            END;
```

In this example, you must keep track of the currently valid variant.

You can define a variant only for the last field in the record. Variant fields can, however, be nested, as in the following example:

```
TYPE Name = PACKED ARRAY [1..20] OF CHAR;
     Date = INTEGER;
     Sex = (Female, Male);
     Hosp = RECORD
            Patient : Name;
            Birthdate : Date;
            Age : INTEGER;
            CASE Parsex : Sex OF

                          Male : ();
                          Female : (CASE Births : BOOLEAN OF
                                     FALSE : ();
                                     TRUE : (Nokids : INTEGER))
            END;
```

This record type contains the name, birthdate, age, and sex of all patients. In addition, it includes a variant field for each woman based on whether she has had any children. A second variant, which contains the number of children, is defined for women who have given birth.

2.3.2.2 Assigning Values to Records - To assign values to a record variable, use an assignment statement to specify a value for each field of the record. The following example shows the declaration of a record and the assignment of values to two of the fields.

```
TYPE ROSTER  = RECORD
               Name: PACKED ARRAY[1..30] OF CHAR;
               Number: INTEGER;
               Grade: REAL;
               END;

VAR Student  : ROSTER;
         Test1,Test2,Test3 : INTEGER;

BEGIN
      WRITELN ('Enter your name.');
      READLN (Student.Name);
      Student.Grade := (Test1 + Test2 + Test3) / 3;
      .
      .
      .
END;
```

If you are initializing a record with a variant clause, you must always specify a value for the tag field, even if it has no tag name. Specifying a value for the tag field, ensures that PASCAL initializes the correct variant, for example:

```
TYPE Sex = (Male, Female);
     Person = RECORD
              Birthdate: RECORD
                         Month: 1..12;
                         Day:   1..31;
                         Year:  INTEGER;
                         END;
              Age: INTEGER;
              CASE sex of
                   Male:   (Bearded:BOOLEAN);
                   Female: (N_Children:INTEGER);
              END;
VAR Dad: Person := ((5, 15, 1921), 62, Male, TRUE);
```

2.3.2.3 Record Type Compatibility - Two records are compatible if their types are identical or equivalent, for example:

```
TYPE Life = RECORD
            Born : INTEGER;
            Died : INTEGER
            END;
     Plantlife = Life;

VAR  Mom, Dad : Life;
     Coleus : Plantlife;
```

The record variables Mom, Dad, and Coleus are all compatible. Mom and Dad are both of type Life, which is equivalent to type Plantlife.

Records of differing types are compatible if they meet the following criteria:

- They have the same number of fields.
- Corresponding field types are compatible.
- Both are packed, or neither is packed. If the types are packed, corresponding fields of subrange types must have equal bounds.

The following type is also compatible with Life and Plantlife:

```
TYPE Coords = RECORD
              X : INTEGER;
              Y : 0..100
              END;
```

The integer subrange 0..100 is compatible with the type INTEGER. However, PASCAL does not check for valid assignments to fields of subrange types. If you assign a record of type Life to a record of type Coord, you must ensure that the value of the field Died is within the subrange 0..100. An out-of-range assignment does not result in an error message.

If the records have variants, these criteria also apply:

- The records must have the same number of variants.
- Corresponding variants must have the same number of fields.
- Corresponding field types within corresponding variants must be compatible.
- The case labels associated with the variants must agree in number, but need not agree in value.
- Corresponding variants in structurally compatible records must have identical tag constant values.
- The tag constant lists in each record must be identical.

For example, assume the program includes the following TYPE definition:

```
TYPE Lets = 'A'..'D';
     Info = RECORD
              Size : INTEGER;
              Calories : INTEGER;
              Protein : 0..40;
              Carb : INTEGER;
              CASE Vits : Lets OF
                   'A','C','D' : ( );
                   'B' : (Niacin, Thiamine : BOOLEAN)
              END;
     Grades = 'A'..'F';
     School = RECORD
                Studentno : INTEGER;
                Class : 1..5;
                Hours : 1..30;
                Incompletes : 1..6;
                CASE Average : Grades OF
                     'B','C','D' : ( );
                     'A' : (Sendlet, Firstsem : BOOLEAN)
                END;
```

The types Info and School are compatible. If you assign a variable of one type to the other, however, you must be sure that both contain the same variant.

2.3.2.4 Record Examples -

Example 1

```
TYPE Taxes = RECORD
               Year : INTEGER;
               Gross : REAL;
               Net : REAL;
               Deductions : INTEGER;
               Itemized : BOOLEAN;
               Interest : ARRAY [1..5] OF REAL
               END;

VAR Fed : Taxes := (1981, 25234.12, 18789.00, 4, TRUE,
                    (5 of 0.05));
```

This example declares and initializes the record Fed of type Taxes.

Example 2

```
     TYPE String = PACKED ARRAY [1..20] OF CHAR;
          Personal = RECORD
                       Name : String;
                       Address : RECORD
                                   Number : INTEGER;
                                   Street, Town : String;
                                   Zip : 0..99999
                                   END;
                       Age : 0..150
                       END;

     VAR Faculty, Mascot, Student : Personal;

     BEGIN
        Faculty.Name := 'Niklaus Wirth        ';
        Faculty.Address.Number := 5;
        Faculty.Address.Street := 'Clausiusstrasse     ';
       .
       .
       .
      END;
```

The type Personal contains the field Address, which is of a record type. To assign values to each of the fields of the record Address, you must also specify the record Faculty in the assignment statement.

2.3.3 Set Types

A set is a collection of data items of the same scalar type, which is known as the base type of the set. Unlike arrays and records, elements in the set cannot be accessed individually. In PASCAL, you use a set as an individual unit. The type definition specifies the values for each element in the set:

Format

```
     TYPE identifier = SET OF base type
```

where:

identifier	specifies the type identifier for the set.
base type	specifies the data type. Each element in the set must be of this data type. You can use the identifier or definition of any scalar type except a real TYPE.

A set can have up to 256 members, and the value of each member must be between 0 and 255. Therefore, real numbers or integers outside the range 0 to 255 cannot be set elements.

After defining a base type, you can declare set variables of that type, for example:

```
     TYPE Sports_Equip = SET OF (Racquet, Shoes, Balls, Boots,
                                 Skis, Poles, Goggles, Swimsuit);

     VAR Ski_Equip,Tennis_Equip,Swim_Equip,Sleep_Equip : Sports_Equip;
```

Sets are compatible if their base types are identical or equivalent, for example:

```
     TYPE Vitamins = SET OF (A, B1, B2, B6, B12, C, D, E, K);
          Nutrients = Vitamins;

     VAR  Watersoluble, Fatsoluble : Vitamins;
          Deficient : Nutrients;
```

The VAR section specifies three mutually compatible sets. Sets with compatible base types are also compatible, for example:

```
     VAR  ASCII : SET OF CHAR;
          Specials : SET OF '!'..'/';
```

These two sets are compatible because the base type of Specials is compatible with the ASCII character set.

Packing has no effect on set compatibility except when passing sets as VAR parameters. An unpacked set is compatible with a packed set if both sets meet the criteria above.

You can build set expressions by using the set operators described in Chapter 3. Set operators allow you to specify set intersection, difference, union, inclusion, and containment. In addition, you can assign a set expression to a set variable. The base type of the variable must include all members of the set to which the expression evaluates.

Example 1

```
     TYPE Caps = SET OF CHAR;
          VAR Vowel : Caps :=  ['A','E','I','O','U'];
              Consonant : Caps := ['B'..'D','F'..'H','J'..'N',
                                   'P'..'T','V'..'Z'];
```

These declarations specify the set type Caps and two set variables, Vowel and Consonant. The set Vowel is initialized with the set of vowel characters as initial values. The set Consonant is initialized with the set of consonants.

Example 2

```
     VAR Ages : SET OF 1..70 := [5,10,15,20,25,30,35,40,45,50,55,60];
         .
         .
         .
```

This example declares and initializes a set with an integer base type.

2.3.4 File Types

A file is a sequence of data components of the same type. The number of components in a file is not fixed; a file can be of any length. The file type definition specifies the type of the file components.

Format

```
     TYPE identifier = FILE OF component type
```

where:

component type	specifies the type of the components of the file. The component type can be any scalar or structured type except a file type or an array or record type containing a file element or field.

The arithmetic, relational, Boolean, and assignment operators do not work on file variables or structures containing file components. For example, you cannot assign one file variable to another file variable, nor can you initialize a file variable.

Type compatibility for files applies only to file parameters. Two file parameters are compatible if their components are compatible and if both are packed or neither is packed. You can pass a file only as a VAR parameter.

PASCAL automatically creates a buffer variable for each file variable you declare. The type of the buffer variable is the same as the type of the file components. To denote the buffer variable, specify the name of the associated file variable followed by a circumflex (^), for example:

```
     TYPE Scores = FILE OF INTEGER;
     VAR Math_Scores : Scores;
```

PASCAL creates Math_Scores^ as an integer buffer variable associated with the file Math_Scores. The buffer variable takes on the value of the file at the current file position. The predeclared input and output procedures move the file position, thus changing the value of the buffer variable.

Example 1

```
     VAR Truthvals : FILE OF BOOLEAN;
```

This declaration specifies a file of Boolean values. The buffer variable for this file is denoted by Truthvals^.

Example 2

```
     TYPE Names = PACKED ARRAY [1..20] OF CHAR;
          Data_File = FILE OF Names;
     VAR Accept_List, Reject_List, Wait_List : Data_File;
```

This example defines the array type Names and the file type DataFile, which contains a list of names. The VAR section specifies three file variables of type DataFile, with associated buffer variables Accept_List^, Reject_List^, and Wait_List^.

Example 3

```
     VAR RESULTS : FILE OF RECORD
                   Trial  : INTEGER;
                   Date   : RECORD
                            Month : (Jan, Feb, Mar, Apr, May, Jun,
                                     Jul, Aug, Sep, Oct, Nov, Dec);
                            Day : 1..31;
                            Year : INTEGER
                            END;
                   Temp, Pressure : INTEGER;
                   Yield, Purity : REAL
                   END;
```

The VAR Declaration specifies a file of records. To access the fields of the record components, you specify Results^.Trial, Results^.Date.Month, and so on.

2.3.4.1 Internal and External Files - A file that is local to a program or subprogram is called an internal file. You can use an internal file only within the scope of the program or subprogram in which it is declared. The system retains an internal file only during execution of the declaring program or subprogram. After execution the file is no longer accessible. The system creates a new file variable with the same name the next time it executes the declaring unit. The contents of the old file are not available. Internal files are not specified in the program heading. Only internal files can be components of structured types.

An external file exists outside the scope of the program in which it is declared. An external file can be created by the current PASCAL program, another PASCAL program, or a program written in another language. The system retains the contents of external file variables after the execution of the program. You must specify the names of external file variables in the program heading. External files cannot be part of a structured type.

2.3.4.2 Text Files - A text file is a file with components of type CHAR. PASCAL defines a file type called TEXT. To declare a text file, specify a variable of type TEXT, for example:

```
     VAR Poem :  TEXT;
```

The text file variable POEM is a file of characters. Text files are divided into lines. Each line ends with a line-separator character. You cannot use this character directly, but you can refer to it indirectly through the predeclared procedures READLN and WRITELN and the predeclared function EOLN.

The predeclared file variables INPUT and OUTPUT are files of type TEXT. These files are the defaults for all the predeclared text file procedures described in Chapter 7. Note that TEXT is not equivalent to FILE OF CHAR.

Example

```
     VAR Guide, Manual :  TEXT;
```

This example declares the Variables Guide and Manual as text files.

2.4 POINTER TYPES

Normally, variables have the same lifetime as the program or subprogram in which they are declared. Program-level variables are allocated in static storage, and subprogram-level variables are allocated on the stack. Some applications, however, require different lifetimes or an unknown number of variables of a certain type. PASCAL allows you to use dynamic variables to fill these requirements.

Dynamic variables are dynamically allocated as needed during program execution. Unlike other variables, dynamic variables are not named by identifiers. Instead, you must refer to them indirectly with pointers.

A pointer type thus allows you to declare any number of pointer variables to refer to dynamic variables of a specified type. Each pointer variable assumes as its value the address of a dynamic variable.

The pointer type definition specifies the type of the dynamic variable to which pointers of the pointer type refer.

Format

```
     TYPE identifier = ^base-type identifier
```

where:

base-type identifier	indicates the type of the dynamic variable to which the pointer type refers. The base type can be any type.

Note the following example:

```
     TYPE Myrec = RECORD
                  A,B,C : INTEGER
                  END;
          Ptr_To_Myrec = ^Myrec;
     VAR M : Ptr_To_Myrec;
```

Variables of a pointer type point to variables of the base type, and are said to be bound to that type. To indicate a pointer variable, specify its name. To indicate the dynamic variable to which a pointer is bound, specify the pointer name followed by a circumflex(^). For example, M is a pointer variable bound to records of type Myrec. Specify M^ to denote the record variable to which M points.

Pointer type definitions are the only place in a PASCAL program where you can use an identifier before you define it. PASCAL allows you to use the base type identifier in a pointer type definition before you define the base type, for example:

```
TYPE Ptr_To_Movie = ^Movie;
     Name = PACKED ARRAY [1..20] OF CHAR;
     Movie = RECORD
               Title, Director : Name;
               Year : INTEGER;
               Stars : FILE OF Name;
               Next : Ptr_To_Movie
               END;
```

The TYPE section specifies the type identifier Movie before defining the type Movie.

The value of a pointer is the storage address of the variable to which it points. Thus, in the example above, the value of the field Next is a pointer to (or address of) a dynamic record variable of type Movie.

Pointers assume values at initialization, by assignment, and through the NEW procedure. The value of a pointer can be any legal storage address. The value NIL indicates that the pointer does not currently specify an address. Thus, a NIL pointer does not point to a variable.

PASCAL allows you to define pointers to types containing files, for example:

```
TYPE X= ^Y;
     Y= RECORD
          P : INTEGER;
          Q : ARRAY[1..3] OF TEXT
          END;
VAR M:X;
```

The pointer type X points to record type Y, which contains a file component in field Q. The files denoted by Q are never closed until execution of the program terminates, unless you use the CLOSE procedure. For example, to close the files defined in the TYPE section above, you must call CLOSE with the parameters M^.Q[1], M^.Q[2], M^.Q[3].

You can assign the constant NIL to a pointer as follows:

```
M := NIL;
```

As a result of the assignment, the pointer variable M does not point to a variable. NIL is the only value you can specify to initialize a pointer.

Example

```
     TYPE Name = ARRAY [1..30] OF CHAR;
          Ptr_To_Hits = ^Hits;
          Hits = RECORD
                 Title, Artist, Composer : Name;
                 Weeks_On_Chart, N_Sold : INTEGER;
                 First_Version : BOOLEAN
                 END;

          VAR Topten : ARRAY [1..10] OF Ptr_To_Hits;
              I : INTEGER;
     BEGIN
        FOR I := 1 TO 10 DO
            Topten[I] := NIL;
        .
        .
        .
     END;
```

This example defines the record type Hits to which pointers of type Ptr_To_Hits refer. The array variable Topten has elements of the pointer type Ptr_To_Hits. Each element of the array is assigned the constant value NIL. The array Topten could be used in creating a list of ten records of type Hits.

2.5 PACKED STRUCTURED TYPES

You can pack any of the structured types by specifying PACKED in the type or variable declaration. Packed data items are stored as densely as possible.

Format

```
     PACKED type definition
```

where:

type definition	defines an array, record, set, or file type.

You can initialize all packed structures in the VALUE or VAR section in the same way that you initialize an unpacked structure of the same type. In general, packed data items require less storage space than unpacked data items of the same type. However, execution is usually slower with packed data items.

In PASCAL, a packed array of characters specifies a string variable.

Example 1

```
     TYPE Ranges = PACKED RECORD
                   Word : 0..65535;
                   Byte : 0..32767;
                   Bit : BOOLEAN
                   END;
```

This example defines a record type with three fields, each of which is packed as densely as possible.

Example 2

```
    VAR City_Census : PACKED ARRAY [1..25] OF 2500..50000;
            I : INTEGER;

        BEGIN
           FOR I := 1 TO 25 DO
              City_Census[I] := 0;
        .
        .
        .
        END;
```

This example declares the variable City_Census as a 25-element array of integer values in the subrange from 2500 through 50000. A value of 0 is assigned to each element of the array.

2.6 TYPE COMPATIBILITY

Type compatibility rules determine the operations and assignments that you can perform with data items of different types. Two scalar types are compatible if their type identifiers are declared equivalent in the TYPE section. In addition, a subrange type is compatible with its base type, and two subranges of the same base type (or equivalent base types) are compatible.

For structured and pointer types, PASCAL enforces structural compatibility. Two structured (that is, arrays, records, files, and sets) or pointer types are compatible if their structures are identical.

The way PASCAL determines structural compatibility depends on the types involved. For instance, the requirements for record compatibility differ from those for array compatibility.

PASCAL uses compatibility rules in the following three contexts:

1. Expression compatibility

2. Assignment compatibility

3. Formal and actual parameter compatibility

Expression compatibility determines the types of operands you can use in an expression. See Chapter 3 for information on expressions.

Assignment compatibility determines the types of values you can assign to variables of each type. Assignment compatibility rules apply to value initializations, assignment statements, and value parameters. Assignment compatibility is described with the assignment statement in Section 5.2.

Formal and actual parameter compatibility determines the types of data you can pass in a parameter list. Value parameters follow the rules for assignment compatibility. Variable parameters follow somewhat different rules. Value and variable parameters are described in Chapter 6.

CHAPTER 3

EXPRESSIONS

An expression is a symbol or group of symbols that PASCAL can evaluate. These symbols can be constants, variables, or functions, or any combination of constants, variables, and functions, combined with operators. The simplest expression is a single variable or constant.

This chapter lists the various operators that PASCAL provides along with the rules for forming arithmetic, relational, logical, and set expressions.

3.1 OPERATORS

PASCAL provides the following types of operators:

- Arithmetic operators (such as +, -, /)
- Relational operators (such as <, >, =)
- Logical operators (such as AND, OR, NOT)
- Set operators (such as IN)

3.1.1 Arithmetic Expressions

An arithmetic expression usually provides a formula for calculating a value. To construct an arithmetic expression, you combine numeric constants, variables, and function identifiers with one or more of the operators from Table 3-1.

Table 3-1: Arithmetic Operators

Operator	**Example**	**Meaning**
+	A+B	Add A and B
-	A-B	Subtract B from A
*	A*B	Multiply A by B
**	A**B	Raise A to the power of B
/	A/B	Divide A by B
DIV	A DIV B	Divide A by B and truncate the result
MOD	A MOD B	Produce the remainder after dividing A by B; B must be greater than 0

The addition, subtraction, multiplication, and exponentiation (+, -, *, and **) operators work on both integer and real values. They produce real results when applied to real values and integer results when applied to integer values. If the expression contains values of both types, the result is a real number. The only exception to these rules concerns exponentiation. PASCAL defines the results of an integer raised to the power of a negative integer as follows:

Base	Exponent	Result
0	Negative or 0	Error
1	Negative	1
-1	Negative and odd	-1
-1	Negative and even	1
Any other integer	Negative	0

For example, the expression 1**(-3) equals 1; (-1)**(-3) equals -1; (-1)**(-4) equals 1; and 3**(-3) equals 0.

The division (/) operator can be used on both real and integer values, but always produces a real result. Use of the division (/) operator can therefore cause errors in precision in expressions involving integers.

The DIV, MOD, and REM operators apply to integer values only.

DIV divides one integer by another, producing an integer result. DIV truncates the result; that is, it drops any fraction. It does not round the result. For example, the expression 23 DIV 12 equals 1, and (-5) DIV 3 equals -1.

The MOD and REM operators return the remainder after dividing one operand by another. Both operators can be used only with integer values and always produce integer results.

The MOD operator can be used only when the divisor is greater than 0; it always returns a positive result. For example, the expression 5 MOD 3 (5 modulo 3) returns a value of 2, and (-5) MOD 3 returns a value of 1.

The REM operator can be used on integers of all sizes and retains the sign of the dividend. For example, the expression 5 REM 3 returns a value of 2, the expression (-5) REM 3 returns a value of -2; and the expression 5 REM (-3) returns a value of 2.

In arithmetic expressions, PASCAL allows you to mix integers, real numbers (single and double precision), and integer subranges. When you assign the value of an expression to a variable, you must ensure that the types of the variable and the expression are compatible. In general, you can assign an integer expression to a real variable. However, you cannot assign a real expression to an integer variable.

Table 3-2 lists the type of the result for all possible combinations of arithmetic operators and operands.

Table 3-2: Result Types for Arithmetic Expressions

Operator (Operation)	**Type of First Operand**	**Type of Second Operand**	**Type of Result**
**	INTEGER	INTEGER	INTEGER
(exponentiation)	INTEGER, REAL	REAL, DOUBLE	REAL
	DOUBLE	INTEGER, REAL, DOUBLE	DOUBLE
*	INTEGER	INTEGER	INTEGER
(multiplication)	INTEGER	REAL	REAL
	REAL	INTEGER, REAL	REAL
	DOUBLE	INTEGER, REAL, DOUBLE	DOUBLE
	REAL, INTEGER	DOUBLE	DOUBLE
/	REAL, INTEGER	REAL, INTEGER	REAL
(division)	DOUBLE	INTEGER, REAL, DOUBLE	DOUBLE
	REAL, INTEGER	DOUBLE	DOUBLe
DIV, MOD, REM	INTEGER	INTEGER	INTEGER

Table 3-2: Result Types for Arithmetic Expressions (Cont.)

Operator (Operation)	Type of First Operand	Type of Second Operand	Type of Result
(division with truncation, modulus, and remainder)			
+,-	INTEGER	INTEGER	INTEGER
(addition, subtraction)	INTEGER	REAL	REAL
	REAL	REAL, INTEGER	REAL
	DOUBLE	REAL, DOUBLE, INTEGER	DOUBLE
	REAL, INTEGER	DOUBLE	DOUBLE

3.1.2 Relational Expressions

A relational expression or condition tests the relationship between two expressions. A relational expression consists of two scalar or string variables or arithmetic expressions, separated by one of the relational operators listed in Table 3-3.

Table 3-3: Relational Operators

Operator	Example	Meaning
=	A = B	A is equal to B
<>	A <> B	A is not equal to B
>	A > B	A is greater than B
>=	A >= B	A is greater than or equal to B
<	A < B	A is less than B
<=	A <= B	A is less than or equal to B

Note that the two characters in each of the <>, >=, and <= operators must appear in the specified order and cannot be separated by a space.

PASCAL produces a Boolean result when it evaluates a relational expression. Every relational expression therefore evaluates to TRUE or FALSE. For example, the condition 2 < 3 is always TRUE; the condition 2 > 3 is always FALSE.

3.1.3 Logical Expressions

Logical expressions test the truth value of combinations of conditions. A logical expression consists of two or more expressions that have Boolean results, separated by one of the logical operators in Table 3-4.

Table 3-4: Logical Operators

Operator	Example	Result
AND	A AND B	TRUE if both A and B are TRUE
OR	A OR B	TRUE if either A or B is TRUE, or if both are TRUE
NOT	NOT A	TRUE if A is FALSE, and FALSE if A is TRUE

The AND and OR operators combine two conditions to form a compound condition. The NOT operator reverses the truth value of a condition, so that if A is TRUE, then NOT A is FALSE.

As with relational expressions, the result of a logical expression is a Boolean value. Note that the entire logical expression is always evaluated, even if the expression value could be uniquely determined from only a part of the expression.

3.1.4 Set Expressions

You can use the operators in Table 3-5 with set variables and constants.

Table 3-5: Set Operators

Operator	Example	Meaning
+	A+B	Union of sets A and B
*	A*B	Intersection of sets A and B
-	A-B	Set of those elements of A that are not also in B
=	A=B	Set A is equal to set B
<>	A<>B	Set A is not equal to set B
<=	A<=B	Set A is a subset of set B
>=	A>=B	Set B is a subset of set A
IN	A IN B	A is an element of set B

The set operators (+, *, -, =, <>, <=, and >=) require both operands to be set values. The IN operator, however, requires a set expression as its second operand and a scalar expression of the associated base type as its first operand, for example:

```
    2*3 IN [1..10]
```

The value of this expression is TRUE, because 2*3 evaluates to 6, which is a member of the set [1..10].

3.1.5 Precedence of Operators

The operators in an expression establish the order in which PASCAL evaluates the expression. Table 3-6 lists the order of precedence of the operators, from highest to lowest.

Table 3-6: Precedence of Operators

Operators	**Precedence**
NOT	Highest
**	
*, /, DIV, MOD, REM, AND	
*, -, OR	
=, <>, <, <=, >, >=, IN	Lowest

PASCAL evaluates operators of equal precedence (such as + and -) from left to right. You must use parentheses for correct evaluation when you combine relational operators, for example:

```
    A <= X AND B <= Y
```

Without parentheses, PASCAL attempts to evaluate this expression as A<=(X AND B)<=Y and generates an error. The expression needs parentheses, as follows:

```
    (A <= X) AND (B <= Y)
```

To evaluate the rewritten expression, PASCAL compares the truth values of the two relational expressions.

You can use parentheses in any expression to force a particular order of evaluation, for example:

Expression:	Evaluates to:
8 * 5 DIV 2 - 4	16
8 * 5 DIV (2 - 4)	-20

PASCAL evaluates the first expression according to the normal rules for precedence. First, it multiplies 8 by 5 and divides the result (40) by 2. Then, it subtracts 4 to get 16. The parentheses in the second expression, however, force PASCAL to subtract before multiplying or dividing. Hence, it subtracts 4 from 2, getting -2. Then, it divides -2 into 40, with -20 as the result.

Parentheses can also help to clarify an expression. For instance, you could write the first example as follows:

```
((8 * 5) DIV 2) - 4
```

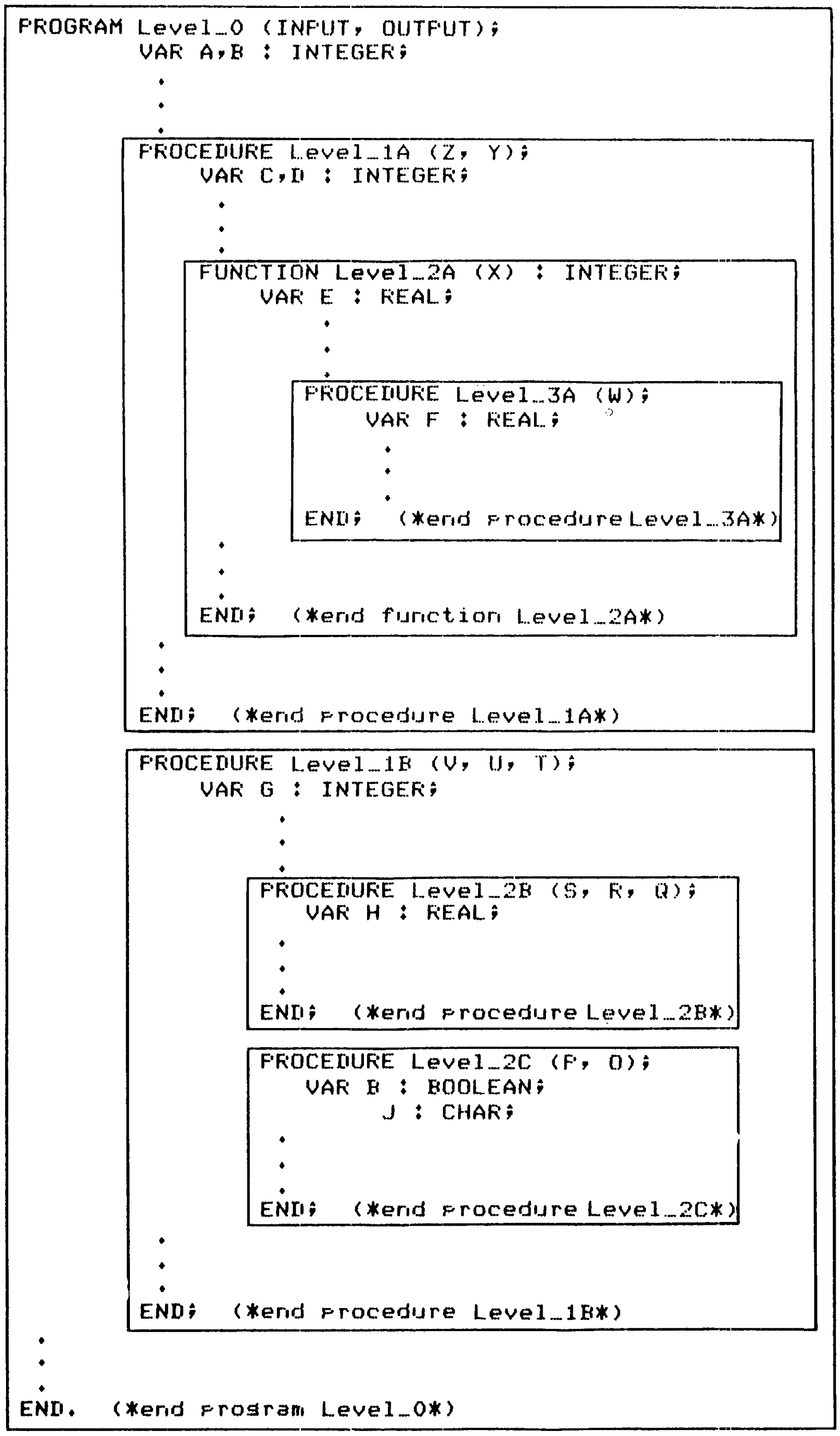
The parentheses eliminate any confusion about how the expression is to be evaluated.

3.2 SCOPE OF IDENTIFIERS

The scope of an identifier is the part of the program in which you have access to the identifier. In a PASCAL program, the scope of a constant, type, variable, or subprogram identifier is the block in which the identifier is declared. Figure 3-1 illustrates the scope of identifiers declared at various levels.

Declarations in the main program block specify global identifiers, which can be accessed in the main program and in all nested subprograms. For example, A and B in Figure 3-1 are global identifiers. They can be accessed from any level in the program.

Declarations in subprogram blocks specify local identifiers. You can use a local identifier in the subprogram that contains its declaration and in all its nested subprograms. For example, the identifiers C and D are local to procedure Level_1A and its nested subprograms Level_2A and Level_3A. You can use C and D in any of these subprograms, but not in the main program or in the subprograms Level_1B, Level_2B, and Level_2C.



```
PROGRAM Level_0 (INPUT, OUTPUT);
        VAR A,B : INTEGER;
            .
            .
            .
        PROCEDURE Level_1A (Z, Y);
            VAR C,D : INTEGER;
                .
                .
                .
            FUNCTION Level_2A (X) : INTEGER;
                VAR E : REAL;
                    .
                    .
                    .
                PROCEDURE Level_3A (W);
                    VAR F : REAL;
                        .
                        .
                        .
                END;  (*end procedure Level_3A*)
                .
                .
                .
            END;  (*end function Level_2A*)
            .
            .
            .
        END;  (*end procedure Level_1A*)
        PROCEDURE Level_1B (V, U, T);
            VAR G : INTEGER;
                .
                .
                .
                PROCEDURE Level_2B (S, R, Q);
                   VAR H : REAL;
                   .
                   .
                   .
                END;  (*end procedure Level_2B*)
                PROCEDURE Level_2C (P, O);
                   VAR B : BOOLEAN;
                         J : CHAR;
                   .
                   .
                   .
                END;  (*end procedure Level_2C*)
            .
            .
            .
        END;  (*end procedure Level_1B*)
.
.
.
END.  (*end program Level_0*)
```

Figure 3-1: Scope of Identifiers

Similarly, local identifiers declared in Level_1B are accessible to Level_1B, Level_2B, and Level_2C, but not to Level_1A, Level_2A, Level_3A, or the main program.

In general, once you define an identifier, it retains its meaning within the block containing its declaration. You can, however, redefine an identifier in a subprogram at a lower level. If you do so, the identifier assumes its new meaning only within the scope of the redefining block. Outside this block, the identifier keeps its original meaning. For example, B is declared at program level and redefined in Level_2C. Within the scope of Level_2C, B denotes a Boolean variable. Everywhere else in the program, however, B denotes an integer.

The identifiers accessible to each routine in Figure 3-1 are listed below.

Routine	Variables
Main program	A, B (integer)
Level_1A	A, B (integer), C, D
Level_2A	A, B (integer), C, D, E
Level_3A	A, B (integer), C, D, E, F
Level_1B	A, B (integer), G
Level_2B	A, B (integer), G, H
Level_2C	A, B (Boolean), J

CHAPTER 4

PROGRAM HEADING AND DECLARATION SECTION

The first two parts of a PASCAL program are the program heading and the declaration section. The program heading specifies the program name and the input and output files.

The declaration section can contain the following sections:

• LABEL	declares labels for use by the GOTO statement
• CONST	defines identifiers to represent constant values
• TYPE	defines user-defined, structured, and pointer types
• VAR	declares and optionally initializes variables of all types
• VALUE	initializes variables
• PROCEDURE and FUNCTION	declare subprograms

Your program need not include all these sections, but the sections that are present must appear in the order listed above. Although you can specify many labels, constants, types, variables, and subprograms, each section can appear only once in each declaration section.

This chapter describes the program heading (Section 4.1), label declarations (Section 4.2), and constant definitions (Section 4.3). It also outlines type definitions (Section 4.4), variable declarations (Section 4.5), and value declarations (Section 4.6). Chapter 6 describes the use of procedures and functions in detail.

4.1 THE PROGRAM HEADING

The program heading begins the PASCAL program. It gives the program a name and lists the external file variables the program uses.

Format

```
[[ [OVERLAID] ]]  PROGRAM program name  [[ (filename [[ ,filename... ]] ) ]] ;
```

where:

[OVERLAID]	specifies that the program can share global variables with separately compiled modules. See Section 6.8.
program name	specifies the name of the program; only the first six characters are significant.
filename	specifies the identifier associated with an external file that the program uses.

The program name appears only in the heading and has no other purpose within the program. The program name cannot be redefined at program level.

The file names listed in the program heading correspond to the external files that the program uses. The heading must include the names of all the external file variables. The predeclared text file variables INPUT and OUTPUT, by default, refer to your terminal (in interactive mode) or the batch input and log files (in batch mode). You must declare file variables for all other external files in the main program declaration section, and specify those variables in the program heading. Refer to Chapter 7 for more information on files.

Example 1

```
PROGRAM Test1;
```

The program heading names the program Test1, but omits the file variable list. This program does not use the terminal or any external file.

Example 2

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

The program heading names the program Squares and specifies the predeclared file variables INPUT and OUTPUT.

Example 3

```
PROGRAM Payroll (Employee, Salary, Output);
```

The program heading names the program Payroll and specifies the external file variables Employee, Salary, and Output.

4.2 LABEL DECLARATIONS

A label makes a statement accessible from a GOTO statement. The label section must list all the labels in the corresponding executable section.

Format

```
LABEL label ⟦,label ...⟧ ;
```

where:

label	specifies an unsigned integer. When you declare more than one label, you can specify the labels in any order.

The label is an unsigned integer. Labels can be listed in any order if more than one label is defined. A label can precede any statement in the program, but can be accessed only by a GOTO statement. Within the program, a colon (:) must be placed between the label and the statement.

The scope of a label is the block in which it is declared. Therefore, you can transfer control from one program unit to another program unit in which the former is nested, for example:

```
PROGRAM Trial (INPUT,OUTPUT);
     LABEL 75;
      .
      .
      .
     PROCEDURE Max;
          LABEL 50;
           .
           .
           .
          BEGIN
               50 : WRITELN ('Testing fairness of tosses');
                .
                .
                .
               GOTO 75;
          END; (*end of procedure Max*)
     BEGIN
           .
           .
           .
          75 : WRITELN ('Not fair! A weighted coin!');
           .
           .
           .
     END.
```

The GOTO statement in the procedure Max transfers control to the main program statement that has the label 75. However, you cannot use a GOTO statement in the main program to transfer control into the procedure at label 50.

Example

```
LABEL 0, 6656, 778, 4352;
```

The label section specifies four labels: 0, 6656, 778, and 4352. Note that the labels need not be specified in numeric order.

4.3 CONSTANT DEFINITIONS

The constant section defines identifiers to represent constant values.

Format

```
    CONST constant name = value;   [[constant name = value;  ...  ]]
```

where:

constant name	specifies the identifier to be used as the name of the constant.
value	specifies an integer, a real number, a string, a Boolean value, or the name of another constant that is already defined. Note that the value assigned to a constant identifier cannot be an expression. String values must be enclosed in apostrophes.

The use of constant identifiers generally makes a program easier to read, understand, and modify. If you need to change the value of a constant, you can simply modify the CONST declaration instead of changing each occurrence of the value in the program. This capability makes programs simpler to maintain and easier to transport.

Example

```
CONST Rain = TRUE;
      Year = 2001;
      Pi = 3.1415927;
      Comma = ',';
      Country = 'United States';
      Citizenship = Country;
```

This CONST section defines six constants. The identifier Rain is equal to the Boolean value TRUE. The identifier Year represents an integer, and Pi represents the real number 3.1415927. The identifier Comma represents a character, and the identifier Country represents a string. Characters and strings must be enclosed in apostrophes in the CONST section. The identifier Citizenship represents the symbolic constant Country and thus represents a character string. Note that, since Citizenship represents a symbolic value and not a string, apostrophes are not used.

4.4 TYPE DEFINITIONS

The type definition introduces the name and the set of values for a type. Chapter 2 describes data types and includes examples of type definitions.

Format

```
     TYPE type identifier = type definition;
        [[ type identifier = type definition;...  ]]
```

where:

type identifier	specifies the identifier to be used as the name of the type.
type definition	defines a type. The types are shown in Figure 4-1.

Note that you can use the identifier for a previously defined type in place of the type definition for a new type. In addition, you can define packed types for arrays, records, sets, and files, as described in Section 2.5.

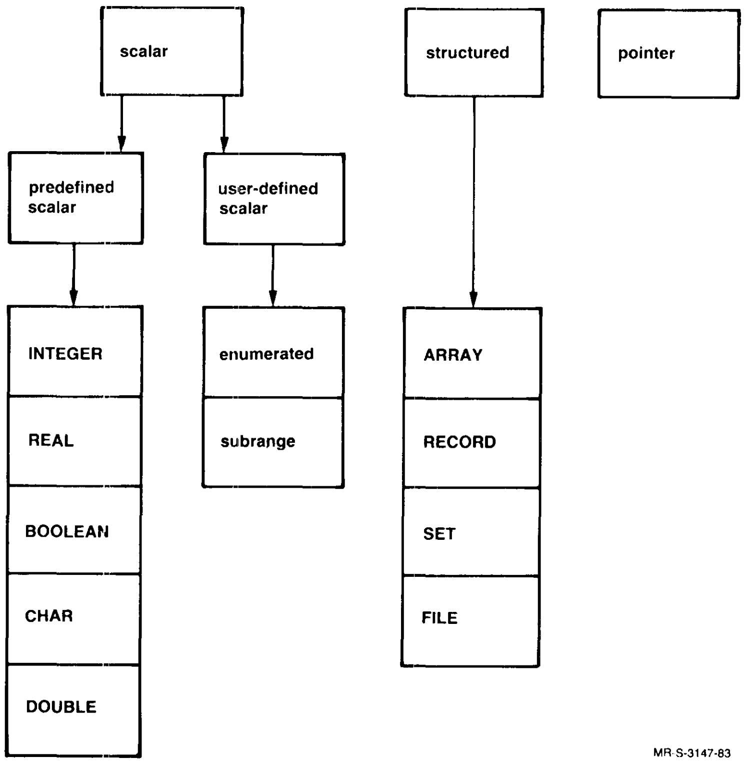


Figure 4-1: PASCAL Data Types

4.5 VARIABLE DECLARATIONS

The variable declaration creates a variable and associates an identifier and a type with the variable. Optionally, the variable declaration can be used to assign an initial value to a variable. Chapter 2 describes data types and shows how to declare and initialize variables of each type.

Format

```
    VAR variable name ⟦,variable name... ⟧  : type  ⟦ := value ⟧ ;
      ⟦variable name ⟦,variable name... ⟧  : type  ⟦ := value ⟧ ;...⟧
```

where:

variable name	specifies the identifier to be used as the name of the variable.
type	names or defines a type. The type can be one of the types shown in Figure 4-1.
value	specifies the initial value associated with the identifier. The value must be of the same data type as the identifier.

You can also declare packed array, record, set, and file variables, as described in Section 2.5. Note, however, that you cannot initialize file variables.

4.6 VALUE DECLARATIONS

The value section initializes variables that are declared in the main program declaration section. You can initialize scalar, array, record, and set variables with constants of the same type as the variable's type.

The description below presents general information on value initializations. The exact format of the value initialization depends on the type of variable being initialized. For detailed formats and examples, refer to the section in Chapter 2 that describes the type of variable you need to initialize.

Format

```
     VALUE variable name := value;
          ⟦variable name := value; ... ⟧
```

where:

variable name	names the variable to be initialized. You cannot specify a list of variable names.
value	specifies a constant of the same type as the variable, or specifies a constructor for an array or record variable.

You must specify a value of the correct type for each variable being initialized. You must not specify an expression. Scalar variables require scalar constants, and set variables require set constants. For arrays and record variables, you specify the value to be assigned to each element or field in a parenthesized list called a constructor. An array or record constructor must contain one constant value of the appropriate type for each component of the structure.

The value initialization can appear only in the main program declaration section. You cannot use a value section in procedures, functions, or modules.

CHAPTER 5

PASCAL STATEMENTS

PASCAL provides several statements to perform the actions within the program. Any of these statements can appear anywhere in the executable part of a PASCAL program, procedure, or function. PASCAL also includes the compound statement, which allows you to group statements.

This chapter presents reference information on each of the statements, organized as follows:

- The compound statement
- The assignment statement
- Conditional statements:

  IF-THEN

  IF-THEN-ELSE

  CASE
- Repetitive statements:

  FOR

  REPEAT

  WHILE
- The WITH statement
- The GOTO statement
- The procedure call

PASCAL includes simple and compound statements. A simple statement can be executed and is complete in itself; that is, it is not made up of other statements. The simple statements are the assignment statement, the GOTO statement, and the procedure call.

A compound statement is an arrangement of simple statements that executes sequentially. You can use a compound statement anywhere in the program that a simple statement is allowed. This manual uses the term statement to mean either a simple or a compound statement.

Simple and compound statements can also be used in structured statements. A structured statement is a group of statements that can be executed either in sequence, conditionally, or repeatedly. The structured statements are the conditional, repetitive, and WITH statements. A compound statement can also be considered a type of structured statement.

5.1 THE COMPOUND STATEMENT

The compound statement allows you to group PASCAL statements for sequential execution as a single statement.

Format

```
     BEGIN
          statement1 ;statement2,...  statement n;
     END;
```

where:

statement	denotes a simple or compound statement.

You can create a compound statement using any combination of PASCAL statements, including other compound statements. You must use semicolons to separate the statements that make up the compound statement; however, no semicolon is required between the last statement and the END delimiter. PASCAL treats the entire compound statement as a simple statement everywhere in the program. Examples of compound statements appear throughout this chapter.

5.2 THE ASSIGNMENT STATEMENT

The assignment statement assigns a value to a variable.

Format

```
     variable name := expression;
```

where:

variable name	specifies the name of a variable of any type (except a file). It could be an array element, a file buffer variable, a function, or a field of a record.
expression	specifies a value, variable name, function reference, Boolean expression, set expression, or arithmetic expression.

Note that the assignment operator is := in PASCAL. Do not confuse this operator with the equal sign (=) operator.

The expression on the right of the operator establishes the value to be assigned to the variable on the left of the operator.

You can use the assignment statement to assign a value to a function identifier or to a variable of any type except a file. The variable and the expression must be of compatible types, with the following two exceptions:

- You can assign an integer expression to a real variable.
- You can assign an integer or single-precision real expression to a double-precision variable.

For structured types, PASCAL enforces structural compatibility in assignments.

Example 1

```
    X := 1;
```

The variable X is assigned the value 1.

Example 2

```
    Temp := Celsius(Fahrenheit);
```

The value returned by the function Celsius is assigned to Temp.

Example 3

```
    T := A < B;
```

The value of the Boolean expression A < B is assigned to T.

Example 4

```
    Vowel_Set := ['A', 'E', 'I', 'O', 'U'];
```

The set Vowel_Set is assigned the string constants shown. The base type of Vowel_Set must include the characters 'A', 'E', 'I', 'O', and 'U'.

Example 5

```
    My_Array[1] := My_Array[7] + Your_Array[14];
```

The first element of My_Array is assigned the sum of the seventh element of My_Array and the fourteenth element of Your_Array.

Example 6

```
    My_Array := Your_Array;
```

The value of each element of the array Your_Array is assigned to the corresponding element of the array My_Array.

Example 7

```
    Awardrec := New_Winner;
```

Assume that Awardrec and New_Winner are record variables of assignment-compatible types. This example assigns the value of each field of New_Winner to the corresponding field of Awardrec.

Example 8

```
     Ages := Ages-[10+7];
```

Assume that the base type of the set variable Ages is the integer subrange 0..255. This example assigns the value of the set expression Ages-[10+7] to the variable Ages.

5.3 CONDITIONAL STATEMENTS

A conditional statement selects a statement for execution depending on the value of an expression. PASCAL provides three conditional statements:

- IF-THEN statement
- IF-THEN-ELSE statement
- CASE statement

These are described in the following sections.

5.3.1 The IF-THEN Statement

The IF-THEN statement causes the conditional execution of a statement.

Format

```
     IF expression THEN statement;
```

where:

expression	specifies a Boolean expression.
statement	indicates a simple or compound statement.

The statement is executed only if the value of the expression is TRUE. If the value of the expression is FALSE, program control passes to the statement following the IF-THEN statement.

The THEN clause can specify a compound statement. However, note that, if you use the compound statement, you must not place a semicolon between the words THEN and BEGIN. The example below shows a semicolon immediately following the word THEN:

```
     IF Day = Thurs THEN;     (* misplaced semicolon *)
     BEGIN
             statement
             .
             .
             .
     END;
```

As a result of the misplaced semicolon, the empty statement becomes the object of the THEN clause. In this example, the compound statement following the IF-THEN statement is executed regardless of the value of Day.

Example 1

```
IF ((X*37/Constant) + Factor) > 1000.0 THEN
     Answer := Answer - Factor;
```

If the value of the arithmetic expression is greater than 1000.0, a new value is assigned to the variable Answer.

Example 2

```
IF (A > B) AND ( B > C) THEN
     D := A - C;
```

If both relational expressions are true, D is assigned the value of A - C.

Example 3

```
IF (Name = 'SMITH') AND (INITIAL = 'J') THEN
     BEGIN
     Count := Count + 1;
     Smithadd[Count] := Address;
     WRITELN ('J SMITH NO.  ',Count, ' LIVES AT ', Address)
     END;
```

This example counts the number of J SMITHs, prints each one's street address, and stores it in an array.

Example 4

```
IF Day = Thurs THEN
   FOR I := 1 TO MaxEmp DO
        Pay[I] := Salary[I] * (1-Tax_Rate-FICA);
```

If the current value of the variable Day is Thurs, the FOR loop is executed, and values for Pay[I] are computed. If the value of Day is not Thurs, the FOR loop is not executed; and program control passes to the statement following the end of the loop.

5.3.2 The IF-THEN-ELSE Statement

The IF-THEN-ELSE statement is an extension of the IF-THEN statement that includes an alternative statement, the ELSE clause. The ELSE clause is executed if the test condition is false.

Format

```
IF expression THEN statement1 ELSE statement2
```

where:

expression	specifies a Boolean expression.
statement1	denotes the statement to be executed if the expression is true.
statement2	denotes the statement to be executed if the expression is false.

The objects of the THEN and ELSE clauses can be any simple or compound statement, including another IF-THEN or IF-THEN-ELSE statement. The ELSE clause always modifies the closest IF-THEN statement.

```
IF A=1 THEN
       IF B<>1 THEN C:=1
          ELSE D:=1;
```

By definition, PASCAL interprets this statement as if it included BEGIN and END delimiters, as follows:

```
IF A=1 THEN
       BEGIN
          IF B<>1 THEN C:=1
          ELSE D:=1
       END;
```

The variable D is assigned the value 1 if both A and B are equal to 1.

Example 1

```
IF Disease THEN
     WRITELN ('This person is sick.')
ELSE WRITELN ('This person is healthy.');
```

This example prints a different line of text depending on the value of the Boolean variable Disease. Note that Disease is a Boolean expression, so you need not specify Disease = TRUE.

Example 2

```
IF Balance < 0.0 THEN
 BEGIN
   WRITELN ('Overdrawn by ',ABS(Balance));
   WRITELN ('Loan of ',Loan,' at ',Rate,
            ' % automatically deposited');
   Balance := Balance + Loan;
   BILL_AMT := Loan * (1+Rate)
 END
ELSE WRITELN ('No Loan issued this month ');
WRITELN ('Balance is ',Balance);
```

If the value of Balance is negative, the compound statement is executed. The compound statement prints two lines of notification, adds Loan to Balance, and computes the amount of the bill for the loan. A zero or positive Balance results in a message stating that no loan was issued. The WRITELN procedure that prints the final Balance is independent of the conditional statement and is always executed.

5.3.3 The CASE Statement

The CASE statement causes one of several statements to be executed, depending on the value of a scalar expression.

Format

```
CASE case selector OF
     case-label list :  statement
      [[;case-label list :  statement...]]
      [[OTHERWISE statement;...]]
END;
```

where:

case selector	specifies an expression that evaluates to any scalar type except a real type.
case-label list	specifies one or more constants of the same type as the case selector, separated by commas.

Each case-label list is associated with a statement that may be executed. The list contains the value of the case-selector expression for which the system executes the associated statement. You can specify the case labels in any order. However, the difference between the largest and smallest labels must not exceed 1000. Each case label can appear only once within a CASE statement, but can appear in other CASE statements.

At run time, the system evaluates the case-selector expression and chooses which statement to execute. If the value of the case-selector expression does not appear in any case-label list, the system executes the statement in the OTHERWISE clause.

If the value of the case-selector expression does not match one of the case labels and you omit the OTHERWISE clause, the status of the CHECK run-time option determines the action that the system takes. Refer to Section 8.4.3 for run-time options. If CHECK is enabled, the system prints an error message and terminates execution. If CHECK is not enabled, execution continues with the statement following the CASE statement.

Example 1

```
CASE Age OF
     5,6 : IF Birth_Month > Sep THEN Grade := 1 ELSE Grade := 0;
       7 : BEGIN
           Grade := 2;
           Reading_Skill := TRUE
           END;
       8 : Grade := 3
END;
```

At run time, the system evaluates Age and executes one of the statements. If Age is not equal to 5, 6, 7, or 8, and the CHECK option is enabled, an error occurs and execution is terminated.

Example 2

```
CASE Age OF
     5,6 : IF Birth_Month > Sep THEN Grade := 1 ELSE Grade := 0;
       7 : BEGIN
           Grade := 2;
           Reading_Skill := TRUE
           END;
       8 : Grade := 3
     OTHERWISE Grade := 0
END;
```

An OTHERWISE clause is added in this example. If the value of Age is not 5, 6, 7, or 8, the value 0 is assigned to the variable Grade.

Example 3

```
     CASE  Alphabetic OF
           'A','E','I','O','U' : Alpha_Flag := Vowel;
                           'Y' : Alpha_Flag := Sometimes
           OTHERWISE Alpha_Flag := Consonant
     END;
```

This example assigns a value to Alpha_Flag depending on the value of the character variable Alphabetic.

5.4 REPETITIVE STATEMENTS

Repetitive statements specify loops, that is, the repetitive execution of one or more statements. PASCAL provides three repetitive statements:

- FOR statement
- REPEAT statement
- WHILE statement

These are described in the following sections.

5.4.1 The FOR Statement

The FOR statement specifies the repetitive execution of a statement based on the value of an automatically incremented or decremented control variable.

Format

```
FOR control variable := initial value {TO    } final value DO statement;
                                      {DOWNTO}
```

where:

control variable	specifies the name of a simple variable of any scalar type except a real type.
initial value	specifies an expression of the same type as the control variable.
final value	specifies an expression of the same type as the control variable.

The control variable, the initial value, and the final value must all be of the same scalar type, but cannot belong to one of the real types. The repeated statement cannot change the value of the control variable. The control variable must be a simple variable; that is, it cannot be an element of an array, a field of a record, the object of a pointer reference, or a file buffer variable.

At run time, completion tests are performed before the statement is executed. In the TO form, if the value of the control variable is less than or equal to the final value, the loop is executed, and the control variable is incremented. When the value of the control variable is greater than the final value, execution of the loop is complete.

In the DOWNTO form, if the value of the control variable is greater than or equal to the final value, the loop is executed, and the control variable is decremented. When the value of the control variable is less than the final value, execution of the loop is complete.

Because completion tests are performed before the statement is executed, some loops are never executed, for example:

```
     FOR Control := N TO N + Q DO
          Week[N] := Week[N] + Newpay;
```

If the value of N + Q is less than the value of N (that is, if Q is negative), the loop is never executed.

When incrementing and decrementing the control variable, PASCAL uses units of the appropriate type. For numeric values, it adds or subtracts one upon each iteration. For values of other types, the control variable takes on each successive value of the type. For example, a control variable of type 'A'..'Z' is incremented (or decremented) by one character each time the loop is executed.

If the FOR loop terminates normally, that is, if the loop exits upon completion and not because of a GOTO statement, the value of the control variable is left undefined. You cannot assume that it retains a value. Therefore, you must assign a new value to the control variable if you use it elsewhere in the program. However, if the FOR loop terminates because of a GOTO statement, it retains the value it had at termination; and you can use that value elsewhere.

Example 1

```
     FOR N := Lowbound TO Highbound DO
          Sum := Sum + IntArray[N];
```

This FOR loop computes the sum of the elements of Int_Array with index values from Lowbound through Highbound.

Example 2

```
     FOR Year := 1899 DOWNTO 1801 DO
          IF (Year MOD 4) = 0 THEN
             WRITELN(Year:4,' IS A LEAF YEAR');
```

The DOWNTO form is used here to print a list of all the leap years in the nineteenth century.

Example 3

```
     FOR I := 1 TO 10 DO
          FOR J := 1 TO 10 DO
             A[I,J] := 0;
```

This example shows how you can directly nest FOR loops. For each value of I, the system steps through all 10 values of J and assigns the value 0 to the appropriate array element.

Example 4

```
     FOR Employee := 1 TO N DO
          BEGIN
            HRS := 40;
            FOR Day := Mon TO Fri DO
                 IF Sick[Employee,Day] THEN
                       HRS := HRS-8;
            Pay[Employee] := Wage[Employee] * HRS
          END;
```

You can combine compound statements as in this example. The inner FOR statement computes the number of hours each employee worked from Monday through Friday. The outer FOR statement resets hours to 40 for each employee and computes each person's pay as the product of wage and hours worked.

5.4.2 The REPEAT Statement

The REPEAT statement groups one or more statements for execution until a specified condition is true.

Format

```
     REPEAT statement  [[ ;statement ...]]  UNTIL expression;
```

where:

expression	specifies a Boolean expression.

Note that the format of the REPEAT statement eliminates the need for a compound statement.

The expression is evaluated after the statements are executed. Therefore, the REPEAT group is always executed at least once.

Example

```
     REPEAT
          READ(X);
          IF (X IN ['0'..'9']) THEN
               BEGIN
               Digit_Count := Digit_Count + 1;
               Digit_Sum := Digit_Sum + ORD(X) - ORD('0')
               END
          ELSE Char_Count := Char_Count+1
     UNTIL EOLN(INPUT);
```

Assume that the variable X is of type Char, and the variables Digit_Count, Digit_Sum, and Char_Count are integers. The example reads a character (X) from the terminal. If X is a digit, the count of digits is incremented by one; and the sum of digits is increased by the value of X. The ORD function is used to compute the value of X. If X is not a digit, the variable Char_Count is incremented by one. The example continues processing characters from the terminal until it reaches an end-of-line (EOLN) condition.

5.4.3 The WHILE Statement

The WHILE statement causes one or more statements to be executed while a specified condition is true.

Format

```
     WHILE expression DO statement;
```

where:

expression	specifies a Boolean expression.

The WHILE statement causes the statement following the word DO to be executed while the expression is true. Unlike the REPEAT statement, the WHILE statement controls the execution of only one statement. Hence, to execute a group of statements repetitively, you must use a compound statement. Otherwise, PASCAL repeats only the single statement immediately following the word DO.

The expression is evaluated before the statement is executed. If the expression is initially false, the statement is never executed. The repeated statement must change the value of the expression. If the value of the expression never changes, the result is an infinite loop.

Example 1

```
     WHILE NOT EOF(File1) DO
           READLN(File1);
```

This statement reads through to the end of the text file File1.

Example 2

```
     WHILE NOT EOLN(INPUT) DO
           BEGIN
               READ(X);
               IF NOT (X IN ['A'..'Z','a'..'z','0'..'9']) THEN
                  Err := Err+1
           END;
```

This example reads an input character from the current line on the terminal. If the character is not a digit or letter, the error count (Err) is incremented by one.

Example 3

```
     Sum := 0;
     Ntests := 1;
     Avg := 100;

     WHILE  (Avg >= 90) AND (Ntests <= Maxtests) DO
             BEGIN
                 Sum := Sum + Test [Ntests];
                 Avg := Sum DIV Ntests;
                 Ntests := NTests +1
             END;
     IF AVG < 90 THEN
     WRITELN ('Your average dropped below 90 as of Test ', Ntests:5);
```

After initializing Sum to zero, this program fragment repeatedly calculates a student's average test score. When the average score falls below 90 (or NTESTS > MAXTESTS), the calculations cease. If the average score is less than 90, the system prints an informative message.

5.5 THE WITH STATEMENT

The WITH statement provides abbreviated notation for references to fields of a record.

Format

```
WITH record variable ⟦,record variable...⟧ DO statement;
```

where:

record variable	specifies the name of the record variable to which the statement refers.

The WITH statement allows you to refer to the fields of a record directly instead of using the record.fieldname format. In effect, the WITH statement opens the scope of the field identifiers so that you can use them as you would use variable identifiers.

Specifying more than one record variable has the same effect as nesting WITH statements. The names must appear in the order of their declaration. Thus, the following two statements are equivalent:

```
WITH Cat, Dog DO
     Bills := Bills + Catvet + Dogvet;
```

and

```
WITH Cat DO
     WITH Dog DO
          Bills := Bills + Catvet + Dogvet;
```

Note that if the record Dog is nested within the record Cat, you must specify Cat before Dog.

Example 1

```
VAR Taxes : RECORD
            Gross : REAL;
            Net : REAL;
            Bracket : REAL;
            Itemized : BOOLEAN;
            Paid : REAL
            END;
    .
    .
    .
WITH Taxes DO
   IF Net < 10000.0 THEN Itemized := TRUE;
```

This statement tests the value of the field Taxes.Net, and sets Taxes.Itemized to TRUE if Taxes.Net is less than 10000.0.

Example 2

```
     TYPE Name = PACKED ARRAY [1..20] OF CHAR;
          Date = RECORD
                 Month : (Jan, Feb, Mar, Apr, May, Jun,
                          Jul, Aug, Sep, Oct, Nov, Dec);
                 Day : 1..31;
                 Year : INTEGER
                 END;

     VAR  Hosp : RECORD
                 Patient : Name;
                 Birthdate : Date;
                 Age : INTEGER
                 END;

       .
       .
       .

     WITH Hosp, Birthdate DO
          BEGIN
            Patient := 'THOMAS JEFFERSON    ';
            Month := Apr;
            Age := 236
               END;
```

The program segment in this example shows how you can use the WITH statement to assign values to the fields of a record. The WITH statement specifies the names of the record variables Hosp and Birthdate. The record names must be in order; that is, Hosp must precede Birthdate. The assignment statements need only specify the field names, for example: Patient instead of Hosp.Patient, Month instead of Hosp.Birthdate.Month, and so forth.

5.6 THE GOTO STATEMENT

The GOTO statement directs the program to exit from a loop or other program segment before its normal termination point.

Format

```
     GOTO label;
```

where:

label	specifies a statement label.

Upon execution of the GOTO statement, program control shifts to the statement with the specified label. The statement can be any PASCAL statement or an empty statement. The label must be defined in the declaration section.

The GOTO statement must be within the scope of the label declaration. In addition, you cannot use a GOTO statement that is outside a compound statement to jump to a label that is within that compound statement.

Example

```
     FOR I := 1 TO 10 DO
          BEGIN
          IF Real_Array[I] = 0.0 THEN
               BEGIN
               Result := 0.0;
               GOTO 10
               END;
          Result := Result + 1.0/Real_Array[I]
          END;

     10:  Invertsum:= Result;
          .
          .
          .
```

This example shows how you can use the GOTO statement to exit from a loop. The loop computes the sum (Invertsum) of the inverses of the elements of Real_Array. If one of the elements is zero however, the sum is set to zero; and the GOTO statement forces an exit from the loop.

5.7 THE PROCEDURE CALL

A procedure call specifies the actual parameters to a procedure and executes the procedure. (See Chapter 6 for a complete description of procedures.)

Format

```
     procedure name  [[(actual parameter  [[,actual parameter...]] )]] ;
```

where:

procedure name	specifies the name of a procedure.
actual parameter	specifies a constant, an expression, the name of a procedure or function, or a variable of any type.

The procedure call associates the actual parameters in the list with the formal parameters in the procedure declaration. It then transfers control to the procedure.

The formal parameter list in the procedure declaration determines the possible contents of the actual parameter list. The actual parameters must be compatible with the formal parameters. Depending on the types of the formal parameters, the actual parameters can be constants, variables, expressions, procedure names, or function names. An unindexed array name in a parameter list refers to the entire array. PASCAL passes actual parameters by the mechanism specified in the procedure declaration.

Example 1

```
     Tollbooth (Change, 0.25, Lane[1]);
```

This statement calls the procedure Tollbooth, specifying the variable Change, the real constant 0.25, and the first element of the array Lane.

Example 2

```
     Taxes (Rate*Income, 'Pay');
```

This statement calls the procedure Taxes, with the expression Rate*Income and the string constant 'Pay' as actual parameters.

Example 3

```
     Halt;
```

This statement calls the predeclared procedure Halt, which has no parameters.

CHAPTER 6

PROCEDURES AND FUNCTIONS

Procedures and functions are program units that perform tasks for other program units. A procedure associates a set of statements with an identifier; the statements are executed as a group. A function returns a value. Each function is associated with a type and an identifier.

Procedures and functions have similar structures and restrictions. This chapter uses the term subprogram in descriptions that apply to both procedures and functions.

PASCAL allows you to use three kinds of subprograms:

- Predeclared subprograms, described in Section 6.1.
- User-declared subprograms written in PASCAL. Sections 6.2 and 6.3 present the general format of subprograms and describe how parameters are passed to subprograms. Sections 6.4 through 6.6 describe how to declare PASCAL procedures and functions.
- External subprograms. This category includes subprograms written in other languages. Sections 6.7 and 6.8 describe external subprograms.

You can include subprograms in the main program compilation unit, or you can compile them separately from the main program in modules. Separately compiled subprograms are considered external to the main PASCAL program, and special usage rules apply (see Section 6.8).

6.1 PREDECLARED SUBPROGRAMS

PASCAL provides predeclared procedures and functions that perform various commonly used tasks, such as input and output operations and mathematical functions. These predeclared subprograms are described in the following sections.

6.1.1 Predeclared Procedures

PASCAL provides procedures that perform input and output, allocate and destroy dynamic variables, supply the system date and time, pack and unpack array variables, and halt program execution. Table 6-1 summarizes these procedures.

Table 6-1: Predeclared Procedures

Procedure	Parameter Type	Action
CLOSE(f)	f = file variable	Closes file f.
DATE(string)	string = variable of type PACKED ARRAY [1..11] OF CHAR]	Assigns current date to string.
DISPOSE(p)	p = pointer variable	Deallocates storage for p^. The pointer variable p becomes undefined.
DISPOSE(p, t1,...,tn)	p = pointer variable t1,...,tn = tag field constants	Releases storage occupied by p^; used when p^ is a record with variants. Tag field values are optional; if specified, they must be identical to those specified when storage was allocated by NEW.
FIND(f,n)	f = file variable n = positive integer expression	Moves the current file position to component n of file f.
GET(f)	f = file variable	Moves the current file position to the next component of f. Then GET(f) assigns the value of that component to f^, the file buffer variable.
HALT	None	Terminates execution of the program.
LINELIMIT(f,n)	f = text file variable n = integer expression	Terminates execution of the program when output to file f exceeds n lines. The value for n is reset to its default after each call to REWRITE for file f.
MARK(a)	a = a variable of type ^INTEGER	Places a marker for use when allocating memory for dynamic variables.
NEW(p)	p = pointer variable	Allocates storage for p^ and assigns its address to p.

Table 6-1: Predeclared Procedures (Cont.)

Procedure	Parameter Type	Action
NEW(p, tl,...,tn)	p = pointer variable tl,...,tn = tag field constants	Allocates storage for p^; used when p^ is a record with variants. The optional parameters tl through tn specify the values for the tag fields of the current variant. All tag field values must be listed in the order in which they were declared. They cannot be changed during execution. NEW does not initialize the tag fields.
OPEN(f, attributes)	f = file variable attributes (see Table 7-2)	Opens the file f with the specified attributes.
PACK(a,i,z)	a = variable of type ARRAY [m..n] OF T i = starting index of array a z = variable of type PACKED ARRAY [u..v] OF T	Moves (v-u+1) elements from array a to array z by assigning elements a[1] through a[1+v-u] to z[u] through z[v]. The upper bound of a must be greater than or equal to (1+v-u).
PAGE(f)	f = text file variable	Skips to the next page of file f. The next line written to f begins on the second line of a new page. The default for f is OUTPUT.
PUT(f)	f = file variable	Writes the value of f^, the file buffer variable, into the file f and moves the current file position to the next component of f.

Table 6-1: Predeclared Procedures (Cont.)

Procedure	Parameter Type	Action
READ(f, v1,...,vn)	f = file variable v1,...,vn = variables	For v1 through vn, READ assigns the next value in the input file f to the variable. You must specify at least one variable (v1). The default for f is INPUT.
READLN(f, v1,...,vn)	f = text file variable v1,...,vn = variables	Performs the READ procedure for v1 through vn, then sets the current file position to the beginning of the next line. The variable list is optional. The default or f is INPUT.
RELEASE(a)	a = a variable of type ^INTEGER	Deallocates memory allocated by the NEW procedure up to the point set by the MARK procedure.
RESET(f)	f = file variable	Enables reading from file f. RESET(f) moves the current file position to the beginning of file f and assigns the first component of f to the file buffer variable, f_^. EOF(f) is set to FALSE unless the file is empty.
REWRITE(f)	f = file variable	Enables writing to file f. REWRITE(f) sets the file f to zero length and sets EOF(f) to TRUE.

UNPACK(z,a,i)	z = variable of type PACKED ARRAY[u..v] OF t
	a = variable of type ARRAY [m..n] OF T
	i = starting index in array a
		Moves (v-u+1) elements from array z to array a by assigning elements z[u] through z[v] to a[i] through a[i+v-u]. The upper bound of a must be greater than or equal to (i+v-u).

Table 6-1: Predeclared Procedures (Cont.)

Procedure	Parameter Type	Action
TIME(string)	string = variable of type PACKED ARRAY [1..11] OF CHAR	Assigns the current time to string.
WRITE(f,p1,...,pn)	f = file variable p1,...,pn = write parameters	Writes the values of p1 through pn into the file f. At least one parameter (p1) must be specified. The default for f is OUTPUT.
WRITELN(f,p1,...,pn)	f = text file variable p1,...,pn = write parameters	Performs the WRITE procedure, then skips to the beginning of the next line. The write parameters are optional. The default for f is OUTPUT.

6.1.1.1 Input/Output Procedures - The PASCAL input and output procedures are:

- CLOSE - Section 7.4
- FIND - Section 7.5
- GET - Section 7.6
- LINELIMIT - Section 7.7
- OPEN - Section 7.8
- PAGE - Section 7.9
- PUT - Section 7.10
- READ - Section 7.11
- READLN - Section 7.12
- RESET - Section 7.13
- REWRITE - Section 7.14
- WRITE - Section 7.15
- WRITELN - Section 7.16

6.1.1.2 Dynamic Allocation Procedures - PASCAL provides the procedures NEW and DISPOSE for use with variables that are dynamically allocated.

<u>NEW</u>

The predeclared procedure NEW allocates memory for a dynamic variable. To refer to the dynamic variable, you must use a pointer variable.

Format

```
    NEW(p);
```

where:

p   specifies a pointer variable.

The NEW procedure sets aside memory for p^, that is, the variable that p points to. The value of p^ is undefined. You cannot assume that the allocated space is initialized.

For example, you declare a pointer variable as follows:

```
    VAR PTR : ^INTEGER;
```

This declares PTR as a pointer to an integer variable. The integer variable and its address, however, do not yet exist. You use the following procedure call to allocate memory for the dynamic variable:

```
    NEW(PTR);
```

This call allocates a variable of type integer. The variable is denoted by PTR^, that is, the pointer variable's name followed by a circumflex(^). This call also assigns the address of the allocated integer to PTR.

<u>DISPOSE</u>

The predeclared procedure DISPOSE deallocates memory for a dynamic variable. As for NEW, you must use a pointer variable to refer to the dynamic variable.

Format

```
    DISPOSE(p);
```

where:

p   specifies a pointer variable.

For example, to deallocate memory for the dynamic variable in the above example, you can issue the following procedure call:

```
    DISPOSE(PTR);
```

As a result of this procedure call, the memory allocated for PTR^ is deallocated; and the variable is destroyed. The value of PTR is now undefined.

Pointer types and dynamic allocation allow you to create linked data structures. An example of the use of pointer types and the NEW and DISPOSE procedures follows.

This program constructs a linked list of records. Each student record contains data on one student, that is, a name and a student ID number. Each record also contains a field that is a pointer to the next record. The program reads a number and a name and assigns each of them to a field of the student record. Then, it inserts the new record on the beginning of the linked list by assigning the "Start" pointer to that new record.

```
PROGRAM LinkedList (INPUT, OUTPUT);

TYPE STUDENT_PTR = ^STUDENT_DATA;
     STRING = PACKED ARRAY[1..20] OF CHAR;
     NUMBER = 1..99999;

     STUDENT_DATA = RECORD
                      Name : STRING;
                      Stud_ID : NUMBER;
                      Next : STUDENT_PTR
                      END;

VAR Start, Student : STUDENT_PTR;
    New_ID : NUMBER;
    New_Name : STRING;
    Count : INTEGER;

PROCEDURE WRITE_DATA(Student : STUDENT_PTR);

(*This procedure prints the list of students.  Because the
printing starts at the beginning of the linked list, the student
names and ID numbers are printed in reverse of the order in
which they were entered.*)

   VAR I,J : INTEGER;
       Next_Student : STUDENT_PTR;
   BEGIN
     WRITELN ('NAME:', 'STUDENT ID#:':29);
     REPEAT
        WRITELN(Student^.Name : 20, Student^.Stud_ID : 7);
        Next_Student := Student^.Next;
        DISPOSE (Student);
        Student := Next_Student
     UNTIL Student = NIL
   END;                                 (*End of WRITE_DATA*)

(* MAIN PROGRAM *)
BEGIN
     Count := 0;
     WRITELN ('Type a 5-digit ID number,'
                   'and a name for each student.');
     WRITELN('Press CTRL/Z when finished.');
     Start := NIL;
     WHILE NOT EOF DO
     BEGIN
        READLN (New_ID, New_Name);
        NEW (Student);
        Student^.Next := Start;
        Student^.Name := New_Name;
        Student^.Stud_ID := NEW_ID;
        Start := Student;
        Count := Count + 1
     END;
     IF Count > 0 THEN
        WRITE_DATA(START)

END.
```

In the main program, the WHILE loop reads a number and a name for one student. The following procedure call allocates memory for a new student record:

```
NEW(Student);
```

The new record is inserted at the beginning of the list, that is, Student^.Next points to the previous head of the list. The value of the new student record is assigned to the Start pointer.

The WRITE_DATA procedure writes the name and student ID number for each student in the linked list. After writing data for one student, the procedure assigns the address of the next record in the list to Next_Student. The following call deallocates memory for one student record:

```
DISPOSE(Student);
```

After deallocating memory, the procedure assigns the value of Next_Student to Student. When the current student record points to NIL, the loop stops executing.

NEW AND DISPOSE -- RECORD-WITH-VARIANTS FORM

You can use the following forms of NEW and DISPOSE when manipulating dynamic variables of a record type with variants:

```
NEW(p,t1,...,tn)

DISPOSE(p,t1,...,tn)
```

The parameter p must be a pointer variable pointing to a record with variants. The optional parameters t1 through tn must be scalar constants. They represent nested tag field values where t1 is the outermost variant.

If you create p without specifying the tag field values, the system allocates enough memory to hold any of the variants in the record. Sometimes, however, a dynamic variable takes values of only a particular variant. If that variant requires less memory than NEW(p) allocates, you should use the NEW(p,t1,...,tn) form.

For example, the following record represents a menu selection:

```
TYPE Menu_Ptr = ^Menu_Order;
     Meat_Type = (Fish, Fowl, Beef);
     Beef_Portion = (Oz_10, Oz_16, Oz_32);
     Menu_Order = RECORD
       CASE Entree : Meat_Type OF
          Fish : (Fish_Type : (Salmon, Cod, Perch, Trout);
                  Lemon : BOOLEAN);
          Fowl : (Fowl_Type : (Chicken, Duck, Goose);
                  Sauce : (Orange, Cherry, Raisin));
          Beef : (Beef_Type : (Steak, Roast, Prime_rib);
                  CASE Size : Beef_Portion OF
                    Oz_10, Oz_16 : (Beef_veg : (Artichoke, Mixed));
                    Oz_32 : (Stomach_Cure : (Bicarbonate,
                                             Antacid,None_Needed)));
       END;

VAR Menu_Selection : Menu_Ptr;
```

You can allocate memory for only the Fish variant as follows:

```
     NEW(Menu_Selection, Fish);
```

The example below shows how to call NEW and specify tag field values for nested variants:

```
     NEW(Menu_Selection, Beef, Oz_32);
```

The tag field values must be listed in the order in which they were declared.

The DISPOSE(p,t1,...,tn) procedure call releases memory occupied by p. The tag field values t1 through tn must be identical to those specified when memory was allocated with NEW, for example:

```
     DISPOSE(Menu_Selection, Beef, Oz_32);
```

This call deallocates the memory allocated by the last NEW procedure call shown above.

6.1.1.3 The MARK and RELEASE Procedures - The MARK and RELEASE procedures give you the opportunity to deallocate memory set aside by several NEW procedures without using several DISPOSE procedures.

If you plan to allocate memory for use by several dynamic variables, you should consider placing a marker at some point. The MARK procedure places an index so that when you use the corresponding RELEASE procedure you deallocate the memory allocated after that marker. This saves you from having to DISPOSE several times, one for each NEW.

The format for MARK is:

Format

```
     MARK(a)
```

where:

a	is a variable of type INTEGER representing a marker.

The format for RELEASE is:

Format

```
     RELEASE(a)
```

where:

a	is the variable specified with MARK.

Note the following example:

```
     MARK(A)
     NEW(Menu_Selection,Fish);
     NEW(Menu_Selection,Beef,Oz_32);
          .
          .
          .
     MARK(B)
     NEW(Menu_Selection,Fowl);
          .
          .
          .
     RELEASE(A)
```

In this example, all three NEW allocations are deallocated by RELEASE(A). If RELEASE(B) had been specified, only the third NEW allocation, NEW(Menu_Selection,Fowl), would have been deallocated.

6.1.1.4 Miscellaneous Predeclared Procedures - PASCAL provides four miscellaneous predeclared procedures: PACK, UNPACK, DATE, and TIME.

The predeclared procedures PACK and UNPACK pack and unpack arrays. Packing means that the data items are stored as densely as possible.

PACK

You can declare arrays to be packed by specifying PACKED in the TYPE or VAR declaration. Sometimes, however, you might want to convert an array to a packed array within the executable section of the program. The predeclared procedure PACK copies elements of an unpacked array to a packed array.

Format

```
     PACK(a,i,z)
```

where:

a	is a variable of type ARRAY [m..n] OF T.
i	is the starting index of a.
z	is a variable of type PACKED ARRAY[u..v] OF T.

The number of elements in a must be greater than or equal to the number of elements in z. PACK(a,i,z) assigns the elements of a, starting with a[i], to the array z, until all of the elements in z are filled. When specifying i, keep in mind that the upper bound of a (that is, n) must be greater than or equal to i+v-u.

For example, you can read integers from a file into an unpacked array, element by element, then pack the whole structure.

```
     VAR A : ARRAY[1..20] OF 0..15;
         P : PACKED ARRAY[1..20] OF 0..15;

     FOR I := 1 TO 20 DO
         READ (A[I]);
     PACK (A,1,P);
```

This program fragment assigns the elements A[1] through A[20] to P[1] through P[20]; that is, all the elements in A are packed into P.

You can pack part of the array A as in the following example.

```
(*Declarations*)

VAR A : ARRAY[1..25] OF 1..15;
    P : PACKED ARRAY[1..20] OF 1..15;

(*Procedure call*)

PACK(A,1,P);
```

This procedure call moves elements of the array A into the packed array P. The parameter 1 specifies that the packing starts with array element A[1]. Thus, the elements A[1] through A[20] are assigned to P[1] through P[20]. Packing need not start with the first element; for example, PACK (A,5,P) packs elements A[5] through A[24] into elements P[1] through P[20].

UNPACK

You can convert a packed array to an unpacked array with the predeclared procedure UNPACK.

Format

```
UNPACK(z,a,i);
```

where:

z	is a variable of type PACKED ARRAY[u..v] OF T.
a	is a variable of type ARRAY[m..n] OF T.
i	is the starting index of a.

For UNPACK, the restrictions on the array indices and the value of i are the same as for PACK.

You cannot pass individual elements of a packed array to a subprogram as a VAR parameter. Therefore, you must unpack the array before you can pass its elements as VAR parameters.

```
(* Declarations *)

VAR P : PACKED ARRAY[1..10] OF CHAR;
    A : ARRAY[1..10] OF CHAR;

PROCEDURE PROCESS_ELEMENTS (VAR CH : CHAR);
    .
    .
    .
END;

(* Part of main program *)

READ (P);
UNPACK(P,A,1);
FOR I := 1 TO 10 DO
   PROCESS_ELEMENTS (A[I]);
```

This program fragment reads characters into the packed array P. The procedure call to UNPACK assigns P[1] through P[10] to the unpacked array elements A[1] through A[10]. Then, for each call to PROCESS_ELEMENTS, one element of A is passed to the procedure.

DATE AND TIME

The predeclared procedures DATE and TIME assign the current date and time to a string variable.

Format

```
DATE(string);
TIME(string);
```

where:

string	specifies a variable of type PACKED ARRAY[1..11] OF CHAR.

The following example demonstrates the use of DATE and TIME:

```
(* Declarations *)

Todays_Date, Current_Time : PACKED ARRAY[1..11] OF CHAR;

(* Procedure calls *)

DATE(Todays_Date);
TIME(Current_Time);
```

These two calls return results in the following format:

```
19-Jan-1980
14:20:25
```

The time is returned in 24-hour format. Thus, the time shown above is 14 hours, 20 minutes, 25 seconds. In the DATE procedure, if the day of the month is a 1-digit number, the leading zero does not appear in the result; that is, a space appears before the date string.

6.1.2 Predeclared Functions

PASCAL provides functions that compute arithmetic values, test certain Boolean conditions, transfer data from one type to another, and perform other miscellaneous calculations. Predeclared functions return a value as a result. Table 6-2 summarizes the predeclared functions.

Arithmetic functions perform mathematical computations. Parameters to these functions can be integer, real, single-precision, or double-precision expressions. The arithmetic functions, except for the absolute and square root functions, return a real value when the parameter is an integer, single-precision, or real value. When the parameter is a double-precision expression, the result is a double-precision value. The absolute and square root functions return a value of the same type as the parameter.

Boolean functions return Boolean values after testing a condition. The EOF and EOLN functions operate on files: EOF tests for the end-of-file condition on a variable of any file type, and EOLN tests for the end-of-line condition on a text file variable. The ODD function tests whether an integer parameter is odd or even. The UNDEFINED function, for which you must supply a real value, returns the value TRUE when the argument is not in normalized floating-point format. Variables containing a value that is not in a normalized floating-point format cause a reserved operand fault when used in arithmetic computations.

Transfer functions take a parameter of one type and return the representation of that parameter in another type. For example, the ROUND function rounds a real number to an integer, and TRUNC truncates a real number to an integer.

The miscellaneous functions include PRED and SUCC. The PRED and SUCC functions operate on parameters of any ordered scalar type (that is, all scalar types except one of the real types). SUCC returns the successor value in the type; PRED returns the predecessor value.

Table 6-2: Predeclared Functions

Category	**Function**	**Parameter Type**	**Result Type**	**Purpose**
Arithmetic	ABS(n)	integer, real, double	same as n	Computes the absolute value of n. The type of n must be either integer, real, or double; and the type of the result is the type of n.
	ARCTAN(n)	integer, real, double	real double	Computes the arctangent of n.
	COS(n)	integer, real, double	real double	Computes the cosine of n.
	EXP(n)	integer, real, double	real double	Computes e**n, the exponential function.
	LN(n)	integer, real, double	real double	Computes the natural logarithm of n. The value of n must be greater than 0.
	SIN(n)	integer, real, double	real double	Computes the sine of n.

Table 6-2: Predeclared Functions (Cont.)

Category	Function	Parameter Type	Result Type	Purpose
	SQR(n)	integer, real, double	same as n	Computes n**2, the square of n. The type of n must be either integer, real, or double; and the type of the result is the type of n.
	SQRT(n)	integer, real, double	real double	Computes the square root of n. If n is less than zero, PASCAL generates an error.
Boolean	EOF(f)	file variable	Boolean	Indicates whether the file position is at the end of the file f. EOF(f) becomes TRUE only when the file position is after the last element in the file. The default for f is INPUT.
	EOLN(f)	text file variable	Boolean	Indicates whether the position of file f is at the end of a line. EOLN(f) is TRUE only when the file position is after the last character in a line, in which case the valur of f^ is a space. The default for f is INPUT.
	ODD(n)	integer	Boolean	Returns TRUE if the integer n is odd; returns FALSE if n is even.
	UNDEFINED(r)	real, double	Boolean	Returns TRUE if the value of r is not in a normalized floating-point format.

Table 6-2: Predeclared Functions (Cont.)

Category	Function	Parameter Type	Result Type	Purpose
Transfer	CARD(s)	set	integer	Returns the number of elements currently belonging to the set s.
	CHR(n)	integer	char	Returns the character whose ordinal number is n (if it exists).
	ORD(n)	any scalar type except a real type	integer	Returns the ordinal (integer) value corresponding to the value of n.
	ROUND(n)	real, double	integer	Rounds the real or double-precision value n to the nearest integer.
	SNGL(d)	double	real	Rounds the double-precision real number d to a single-precision real number.
	TRUNC(n)	real, double	integer	Truncates the real or double-precision value n to an integer.
Miscellaneous	CLOCK	none	integer	Returns an integer value equal to the central processor time used by the current process. The time is not expressed in milliseconds.
	EXPO(r)	real, double	integer	Returns the integer-valued exponent of the binary floating-point representation of r; for example, EXPO(8.0) is 4.

Table 6-2: Predeclared Functions (Cont.)

Category	Function	Parameter Type	Result Type	Purpose
	PRED(a)	any scalar type except a real type	same type as parameter	Returns the predecessor value in the type of a (if a predecessor exists). But it is up to you to make sure there is pred/succ. The compiler will always return with the next ordinal value higher/lower than a. There is no bounds checking. Checking occurs only if the PRED(a) or SUCC(a) is in an assignment statement, for example: (*$CHECK+*) x:=PRED(a)
	SUCC(a)	any scalar type except a real type	same type as parameter	Returns the successor value in the type of a (if a successor exists).

PASCAL also provides additional arithmetic routines available with PASLIB, the PASCAL library. Use the following format to specify a routine from the Common Math Library:

$$\text{FUNCTION routine name (VAR parameter list): } \left\{\begin{array}{l}\text{result type;}\\ \text{FORTRAN}\end{array}\right\}$$

where:

routine name	is one of the names listed in Table 6-3.
parameter list	are acceptable arguments for this routine. See the <u>TOPS-10/TOPS-20 Common Math Library Reference Manual</u>.
result type	is the type of the function result.

Table 6-3 lists these routines and their purpose. For more information concerning these routines, refer to the <u>TOPS-10/TOPS-20 Common Math Library Reference Manual</u>.

Table 6-3: Library Routines

Routine Name	**Purpose**
ARCOS	arc cosine
AINT	truncation to integer
ALOG	natural logarithm
ALOG10	base-10 logarithm
AMAX0	largest of a series
AMIN0	smallest of a series
AMIN1	smallest of a series
AMOD	remainder
ANINT	nearest whole number
ASIN	arc sine
ATAN2	arc tangent of two angles
COSD	cosine (angle in degrees)
COSH	hyperbolic cosine
COTAN	contangent
DABS	double-precision D-floating-point absolute value
DACOS	double-precision D-floating-point arc cosine
DASIN	double-precision D-floating-point arc sine
DATAN	double-precision D-floating-point arc tangent
DATAN2	double-precision D-floating-point arc tangent of two angles
DBLE	conversion from single-precision to double-precision D-floating-point format
DCOS	double precision D-floating-point cosine
DCOSH	double-precision D-floating-point hyperbolic cosine
DCOTAN	double-precision D-floating-point cotangent
DDIM	double-precision D-floating-point positive difference
DEXP	double-precision D-floating-point exponential
DEAXP2.	exponentiation of a double-precision D-floating-point number to the power of an integer

Table 6-3: Library Routines (Cont.)

Routine Name	Purpose
DEAXP3.	exponentiation of a double-precision D-floating-point number to the power or another double-precision D-floating-point number.
DFLOAT	conversion of an integer to double-precision D-floating-point format
DIM	positive difference
DINT	double-precision D-floating-point truncation
DLOG	double-precision D-floating-point natural logarithm
DLOG10	double-precision D-floating-point base-10 logarithm
DMAX1	double-precision D-floating-point largest in a series
DMIN1	double-precision D-floating-point smallest in a series
DMOD	double-precision D-floating-point remainder
DNINT	double-precision D-floating-point nearest whole number
DPROD	double-precision D-floating-point product
DSIGN	double-precision D-floating-point transfer of sign
DSIN	double-precision D-floating-point sine
DSINH	double-precision D-floating-point hyperbolic sine
DSQRT	double-precision D-floating-point square root
DTAN	double-precision D-floating-point tangent
DTANH	double-precision D-floating-point hyperbolic tangent
EXP1.	exponentiation of an integer to the power of another integer
EXP2.	exponentiation of a single-precision number to the power of an integer
EXP3.	exponentiation of a single-precision number to the power of another single-precision number
FLOAT	conversion of an integer to single-precision format
IABS	integer absolute value
IDIM	integer positive difference

Table 6-3: Library Routines (Cont.)

Routine Name	Purpose
IDINT	conversion of a dobule-precision D-floating-point number to integer format
IDNINT	integer nearest whole number for a double-precision D-floating-point number
IFIX	conversion of a single-precision number to integer format
INT	conversion of a single-precision number to integer format
ISIGN	integer transfer of sign
MAX0	largest of a series
MAX1	largest of a series
MIN0	smallest of a series
MIN1	smallest of a series
MOD	integer remainder
MINT	integer nearest whole number for a single-precision number
RAN	random number generator
RANS	random number generator with shuffling
REAL	conversion of an integer to single-precision format
SAVRAN	save teh last random number generated
SETRAN	set the seed value for the random number generator
SIGN	transfer of sign
SIND	sine (angle in degrees)
SINH	hyperbolic singe
SNGL	conversion of a double-precision D-floting-point number to single-precision format
TAN	tangent
TANH	hyperbolic tangent

6.2 FORMAT OF A SUBPROGRAM

Subprograms are similar in format to programs. A subprogram consists of a heading and a block; the block contains a declaration section and an executable section.

The heading specifies the name of the subprogram and lists its formal parameters. For a function, the heading also indicates the type of the value returned. The declaration section defines labels and identifiers for constants, types, variables, procedures, and functions that are used in the subprogram. The executable section contains the statements that perform the actions of the subprogram.

The labels and identifiers declared in the subprogram are local data and are unknown outside the scope of the subprogram. The system does not retain the values of local variables after exiting from the subprogram. The following is a sample subprogram:

```
PROCEDURE PRINT_SYM_ARRAY (VAR A: ARR; Side : INTEGER);

 (* This procedure prints array A if it is symmetric (which is
    determined by the function SYMMETRY). The array is printed
    in row order, one row per line. *)

VAR I, J, K, L : INTEGER;

  FUNCTION SYMMETRY : BOOLEAN;
    (* This function returns true if the array is symmetric; false
       otherwise. *)

  BEGIN
     SYMMETRY := TRUE;
     FOR K := 1 TO Side DO
       FOR L := K TO Side DO
         IF A[K,L] <> A[L,K] THEN SYMMETRY := FALSE;
  END;

                    (* beginning of PRINT_SYM_ARRAY *)

  BEGIN
     IF SYMMETRY THEN
     BEGIN
        WRITELN ('Array entered:');
        FOR I := 1 TO Side DO
        BEGIN
           FOR J := 1 TO Side DO
             WRITE (A[I,J] : 4 );
           WRITELN
        END;
     END
     ELSE WRITELN ('The array is not symmetric.');
  END;
                    (* end of PRINT_SYM_ARRAY body *)
```

Subprograms can be nested within other subprograms. In the previous example, the function SYMMETRY is nested in the procedure PRINT_SYM_ARRAY. The order of nesting determines the scope of an identifier.

Data items declared in any particular block of a PASCAL program are considered global to all its nested subprograms. Thus, data items declared in the main program block are global to all subprograms. A subprogram can access its global identifiers. For example, the function SYMMETRY above has no local variables. It uses the global identifiers K and L, and the parameters A and Side, which are declared in PRINT_SYM_ARRAY.

6.3 PARAMETERS

Subprograms communicate data with the main program and with each other by means of parameters. A subprogram can have any number of parameters, but need not have any at all.

The subprogram heading lists the formal parameters, which specify the type of data that will be passed to the subprogram. For example, the formal parameter list for the procedure PRINT_SYM_ARRAY is the following:

```
    (VAR A : ARR; Side : INTEGER)
```

This list specifies two parameters to be passed to PRINT_SYM_ARRAY: the variable A of the previously defined type ARR, and the value Side of type INTEGER.

Each formal parameter corresponds to an actual parameter, specified in the subprogram call. For example, a valid procedure call to PRINT_SYM_ARRAY is the following:

```
    PRINT_SYM_ARRAY (Current_Arr, Current_Side);
```

This procedure call passes the variable Current_Arr and the value of Current_Side to PRINT_SYM_ARRAY.

The formal parameters are identifiers used in the subprogram; they represent the actual parameters in each subprogram call. You can call a subprogram several times with different actual parameters. At execution, each formal parameter represents the variable or value of the corresponding actual parameter. The formal-value parameters, and the actual parameters to which they correspond, must be of identical types.

6.3.1 Formal Parameters

The formal parameter list specifies the identifier for each parameter and the type of each parameter to be used within the subprogram.

Format

```
    ( [[ mechanism specifier ]] identifier list : type;
     [[ [[ mechanism specifier ]] identifier list : type ...]] );
```

where:

mechanism specifier	indicates how PASCAL passes data to this parameter. The mechanism specifiers for PASCAL subprograms are VAR, PROCEDURE, and FUNCTION.
identifier list	specifies one or more identifiers, separated by commas.
type	specifies the type of the parameters in the list. You can pass values, variables, procedures, and functions to a subprogram written in PASCAL, as described below.

PASCAL provides two methods for passing parameters to PASCAL subprograms.

1. Value -- the value of the actual parameter expression is passed to the subprogram. The subprogram cannot change the actual parameters's value during execution. (Value is the default method.)

2. Variable -- the address of the parameter variable is passed to the subprogram. The subprogram can change the parameter's value. The VAR mechanism specifier indicates that a parameter is to be passed as a variable parameter.

Value parameters pass the value of the actual parameter expression to the subprogram. The subprogram does not change the actual parameter's value during execution. Therefore, after the subprogram executes, the value of the actual parameter is the same as before the execution of the subprogram. The following example shows a formal parameter list that includes two value parameters:

```
    PROCEDURE EXAMPLE (Counter :  INTEGER;  NAME :  CHAR);
```

Variable parameters pass the address of the actual parameter expression to the subprogram. The subprogram can change the actual parameter's value. Therefore, after the subprogram executes, the value of the actual parameter is the value assigned to the corresponding formal parameter within the subprogram. To specify a variable parameter, use the reserved word VAR before the parameter in the formal parameter list. The following example shows the same formal parameter list as in the previous example. In this example, the parameters have been defined to be variable parameters.

```
    PROCEDURE EXAMPLE (VAR Counter :  INTEGER;  VAR Name :  CHAR);
```

6.3.1.1 Value Parameters - By default, PASCAL passes value parameters to PASCAL subprograms. When you specify a value parameter, the formal parameter list does not include the reserved word VAR.

The actual parameter corresponding to a formal value parameter must be a compatible expression. Value parameters follow the rules for assignment compatibility. For example, the following list passes all parameters by value:

```
    (A, B :  INTEGER;  C :  CHAR)
```

The actual parameters corresponding to A and B must be integer expressions. The actual parameter corresponding to C must be a character expression.

If the subprogram changes the value of a value parameter, the change is not reflected in the calling program unit. Thus, if you do not want the value of an actual parameter to change as a result of the execution of a subprogram, you pass it as a value parameter.

6.3.1.2 Variable Parameters - To pass a variable parameter, use the reserved word VAR. You must use the VAR specifier to pass file parameters and to pass actual parameter variables with values that change during execution of the subprogram. The corresponding actual parameter must be a variable; it cannot be a constant or an expression.

When you pass a variable as a variable parameter, the subprogram has direct access to the corresponding actual parameter. Thus, if the subprogram changes the value of the formal parameter, this change is reflected in the actual parameter in the calling program unit.

In the example procedure PRINT_SYM_ARRAY, the actual parameter corresponding to A is passed using variable semantics. It must be a variable of the previously defined type, ARR. The actual parameter corresponding to Side is passed by value and must be an integer expression.

The VAR specifier must precede each identifier list that is to be passed using variable semantics. Thus, VAR can appear more than once in the formal parameter list, for example:

```
    (VAR SEA, BREEZE : REAL; WIND : INTEGER; VAR SICK : MED_FILE)
```

As a result of this formal parameter list, the actual parameters corresponding to SEA, BREEZE, and SICK are passed as variable parameters; and the actual parameter corresponding to WIND is passed as a value parameter (the default).

Compatibility

Variables passed to a subprogram as actual variable parameters must be of the same type as the corresponding formal parameters.

The following restrictions also apply to variable parameters:

- You cannot pass an element of a packed structure with the VAR specifier, although you can pass the entire structure. You must unpack the structure or assign its elements to simple variables before you can pass individual elements.
- You cannot pass a variable of a packed set type to a formal parameter that is an unpacked set type, and vice versa.
- You cannot pass a tag field of a record with the VAR specifier (see Section 2.3.2.1). You can pass the entire record or assign the tag field to another variable in order to pass it.

6.3.1.3 Formal Procedure and Function Parameters - PASCAL allows procedures and functions to be passed as parameters to other procedures or functions. To do this, a full procedure or function heading is given as one parameter of the procedure being declared. For example, the following procedure declaration specifies one value parameter and one function parameter:

```
PROCEDURE ACTUAL (VAL: INTEGER;
                  FUNCTION FORMAL (F1:INTEGER):INTEGER);
          .
          .
          .
VAL := FORMAL (VAL + 1);
          .
          .
          .
```

The formal function parameter takes one value integer parameter, and returns an integer value. When procedure ACTUAL is called, you need to supply it with two parameters: an integer value; and, the name of a function which takes one integer argument and returns an integer value. Note that ONLY the function name is passed to the procedure; do NOT supply the function's parameters in the procedure call. They are supplied when the procedure calls the function.

The parameter list of the formal procedure or function may consist of anything that can be defined in a normal procedure or function declaration, including value parameters, VAR parameters, conformant arrays, and even other procedures and functions. These procedures and functions obey the same rules as the formal procedure or function of which they are a parameter.

The following is an example of a procedure heading with formal procedures nested to two levels:

```
PROCEDURE OUTER(PROCEDURE FORMAL1(FUNCTION FORMAL2:REAL; B:REAL));
```

Procedure OUTER has one parameter, a procedure. This procedure has two parameters, a real-typed function (with no parameters) and a real value parameter.

When a subprogram is called with a procedure or function parameter, the parameter lists of the formal and actual parameters must be congruous. This means that the parameter lists must have the same number of parameters; and each corresponding parameter must be of the same kind (value, VAR, etc.) and of the same type. In the following example, procedure YOU_BET could be passed as a parameter to procedure OUTER above, because YOU_BET and FORMAL1 have congruous parameter lists.

```
PROCEDURE YOU_BET(FUNCTION YOU: REAL; BET: REAL);
```

In the following example, procedures PRINTHEX and PRINTOCTAL have congruous parameter lists. Procedure PRINTBINARY's parameter list is not congruous to either of the others because both of its parameters are value parameters.

```
PROCEDURE PRINTHEX (VAL: INTEGER; VAR SIZE: INTEGER);

PROCEDURE PRINTOCTAL (I: INTEGER; VAR F: INTEGER);

PROCEDURE PRINTBINARY (NUM: INTEGER; WIDTH: INTEGER);
```

An optional syntax for formal procedure and function parameters is to omit the parameter list in the declaration. If this is done, no checking on the number or type of parameters is possible, and only value parameters are allowed when calling the formal procedure or function. No conformant array, procedure, or function parameters can be passed to a formal procedure or function unless they are explicitly declared.

For information on calling sequences for user-defined functions and procedures, refer to Appendix G.

6.3.2 Conformant Arrays

Some programming applications require general subprograms that can process arrays of varying size. PASCAL allows you to declare such subprograms using conformant arrays. You can call the subprogram with arrays of different sizes, as long as their bounds are within those specified by the formal parameter.

For example, you could write a procedure that sums the components of a one-dimensional array. Each time you use the procedure, however, you might want to pass arrays with different bounds. Instead of declaring multiple procedures using arrays of each possible size, you can use a conformant-array parameter. The procedure treats the formal parameter as if its bounds were those of the actual parameter.

The format of a conformant-array parameter is:

```
     array id : ARRAY[idlow..idup : scalar-type id]
                OF conformant-array type;
```

where:

idlow	is the lower bound constant identifier
idup	is the upper bound constant identifier
array id	is one or more identifiers associated with the array

Idlow and idup are bound identifiers. They define the lower and the upper limits, respectively, of the array. Each id is treated as a constant value in the subprogram; therefore, you cannot assign values to the id other than in the definition. You cannot use a subrange to define the limits.

Scalar type specifies the data type of the index. Note that you must use a type identifier to specify the range of indices. The type identifier can be one of the predefined scalar types INTEGER or CHAR.

Conformant-array type can be either a type identifier or another conformant-array specification.

Multidimensional arrays can also use conformant array parameters. The format is:

```
     array-id : ARRAY [idlow .. idup  : scalar-type id]
     OF ARRAY [idlow .. idup : scalar-type id]
     OF conformant-array type;
```

An abbreviated form can also be used to define multidimensional conformant arrays. The format is:

```
     array-id : ARRAY [idlow .. idup  : scalar-type id;
                       idn.. idn : scalar-type idn; ...] OF
                       conformant-array type;
```

When a subprogram with conformant-array parameters is called, the IDs (the bound identifiers) assume the values of the lower and upper bound values of the actual parameters. These values are those specified in the definition section of the actual-array parameter.

A conformant-array parameter can have up to one conformant dimension packed. This means that a one-dimensional conformant-array parameter can be packed or unpacked. A multidimensional conformant-array parameter can be unpacked, or its last (rightmost) conformant dimension can be packed. Note that this restriction applies only to the conformant part of the parameter; the conformant array type can be of any type, packed or unpacked.

The components and indices of the actual and formal conformant-array parameters must be of compatible types. The rules for conformant-array compatibility are the same as those for other arrays with one exception. That is, the range of the index types of the actual-array parameter must be within the range specified for the formal parameter.

Example 1

The following program shows how to declare and use conformant-array parameters.

```
PROGRAM Dynarr(INPUT, OUTPUT);
(* This program illustrates the use of conformant-array
parameters.
The procedure Sum is called from the main program with two
different actual parameters: Arr1 and Arr2.  *)

TYPE Rng = 1..50;
VAR Arr1 : ARRAY[1..5] OF INTEGER;
    Arr2 : ARRAY[7..20] OF INTEGER;
    K,J : INTEGER;

PROCEDURE Sum (VAR Inarr : ARRAY [Low..High : INTEGER] OF
INTEGER);
(* This procedure accepts actual-array parameters with integer
components whose indices are within the range specified by
Rng. *)

   VAR I,Ans :  INTEGER;
   BEGIN
     Ans := 0;
     FOR I:= Low TO High DO
       Ans := Ans + Inarr[I];
     WRITELN('The sum of the components is: ',Ans)
     END;    (*end Sum*)

(* MAIN PROGRAM *)
BEGIN
     WRITELN ('TYPE 5 INTEGERS');
     FOR K:= 1 TO 5 DO
       READ (Arr1[K]);
     Sum (Arr1);
     WRITELN('TYPE 14 INTEGERS');
     FOR J:= 7 TO 20 DO
       READ(Arr2[J]);
     Sum (Arr2)
END.
```

This program sums the components of the arrays Arrl and Arr2. The procedure Sum includes a 1-dimensional conformant-array parameter, Inarr, whose indices are of type Rng. Within the main program, Sum is called with two different arrays: Arrl with index type [1..5], and Arr2 with index type [7..20].

The first procedure call, Sum(Arrl), passes Arrl to Sum. Low assumes the value of 1, and High assumes the value of 5. The FOR loop processes array components Inarr[1] to Inarr[5]. When Sum is called with Arr2, Low assumes the value of 7, and High assumes the value of 20. When Sum is called with Arr2, the FOR loop processes the components Inarr[7] to Inarr[20].

Example 2

A conformant-array parameter can have more than one dimension, as in this example:

```
     TYPE Level_Range = 1..6;
          Nclasses = 1..8;
          Nstudents = 1..40;
          Names = PACKED ARRAY [1..35] OF CHAR;

     VAR Students : ARRAY [1..6,1..8,1..40] OF Names;

        .
        .
        .

     PROCEDURE Kid_Count (School :  ARRAY [Level_Low..Level_High :
                          Level_Range; Nclasses_Low..Nclasses_High :
                          Nclasses; NstudentbLow..Nstudent_High:
                          : Nstudent] OF Names);

        .
        .
        .
```

This example defines School as a three-dimensional conformant-array parameter. Each array passed to School might contain the names of all the students in a particular elementary school. The indices of the array denote the number of grades in the school, the number of classes at each grade level, and the number of students in each class.

The actual-array parameters can have from one to six grades, one to eight classes at any grade level, and one to forty students in any particular class. Furthermore, the indices of the actual-array parameters must be within the ranges shown in the TYPE section. For example, a school with six grades must use integer indices from one to six. Indices of zero to five, for instance, cannot be used.

6.4 DECLARING A PROCEDURE

A procedure is a group of statements that perform a set of actions. The use of procedures allows you to break a complex program into several units, each of which performs one task. For example, a program that computes social statistics from survey data might contain procedures to input and validate the data, select a random sample, and print results.

To declare a procedure, specify its header and block in the procedure and function section. The header consists of the word PROCEDURE and the procedure name along with any parameters you want to include. The block consists of its own declaration section and the executable section. You can declare a procedure in the main program, in a module, or in another subprogram.

Format 1

```
PROCEDURE procedure-identifier  ⟦(formal parameters)⟧ ;

     label section;
     constant section;
     type section;
     variable section;
     procedure-and-function section;

BEGIN
  statement ⟦; statement...⟧
END;
```

Format 2

```
{⟦ [GLOBAL]  ⟧} PROCEDURE procedure id  ⟦(formal parameters)⟧ ; ⟦FORWARD;⟧
{⟦ [FORTRAN] ⟧}
```

Format 3

```
PROCEDURE procedure id  ⟦(formal parameters)⟧ ; {⟦ EXTERN ⟦AL⟧ ; ⟧}
                                                {⟦ FORTRAN;      ⟧}
```

Format 1 shows the format for a procedure that is included within the program that calls it.

Format 2 shows the format for a procedure that can be called externally, that is, it can be called from another program. The FORWARD declaration permits the use of forward references in the declarations section. Forward declarations are described in Section 6.6. The FORTRAN declaration at the beginning of the line indicates that this procedure can be called externally by a FORTRAN program.

Format 3 shows the format for a procedure that is being called externally. The procedure must be compiled separately from the source program that calls it. Placing the FORTRAN declaration at the end of the line indicates that the procedure being called by PASCAL is a procedure written in the FORTRAN language. Refer to Section 6.7.

procedure id

 specifies the identifier to be used as the name of the procedure.

formal parameters

 contains the names and types of the formal parameters. It optionally can include the reserved words VAR, PROCEDURE, and FUNCTION.

label section

 declares local labels.

constant section

 defines local constant identifiers.

type section

 defines local types.

variable section

declares local variables.

procedure-and-function section

declares local procedures and functions.

statement

specifies an action to be performed by the procedure. A procedure can contain any of the statements described in Chapter 5.

A procedure consists of a heading and a block. The procedure block is similar in structure and contents to the main program block, with the following exceptions:

- The declaration section cannot contain VALUE initializations.
- The procedure block ends with END followed by a semicolon (;), rather than a period (.) as in the main program. The procedure does not have a block if it is a forward declaration or is defined externally (EXTERNAL or FORTRAN).

You must declare all the variables that are local to the procedure, but you should not redeclare the formal parameters or the procedure identifier as variable, type, or constant identifiers.

For the two examples that follow, assume that these declarations have been made:

```
CONST NUMBER = 20;
TYPE Range =  0..100;
     List = ARRAY[1..NUMBER] OF Range;

VAR ARR : List;
    Minimum, Maximum : Range;
    Average : REAL;
```

Example 1

```
PROCEDURE READ_WRITE (VAR A : List);
 VAR I : INTEGER;
 BEGIN
  WRITELN ('Type a list of 20 integers',
                     'in the range of 0 to 100.');
  FOR I := 1 TO Number DO
  BEGIN
     READ(A[I]);
     WRITE(A[I]:7);
     WRITELN
  END
 END;
```

The procedure READ_WRITE reads a list of 20 integers, inserts them into the array A, and writes the array. READ_WRITE uses one variable parameter, the array A.

Given the declaration of ARR, the following is a valid procedure call:

```
READ_WRITE(ARR);
```

As a result of this call, the list of integers is written in the array ARR. The value of this array is then returned to the program unit that called the procedure READ_WRITE.

Example 2

```
PROCEDURE MIN_MAX_AVG (VAR Min, Max : Range;
                       VAR Avg : REAL; A : List);
  VAR Sum, NMax, NMin, J : INTEGER;

  BEGIN
    Max := A[1];  Min := Max;  Sum := Max;
    NMax := 1;  NMin := 1;
    FOR J := 2 TO NUMBER DO
    BEGIN
       Sum := Sum + A[J];
       IF A[J] > Max THEN
       BEGIN Max := A[J];
             NMax := 1
       END
       ELSE IF A[J] = Max THEN
             NMax := NMax + 1;
       IF A[J] < Min THEN
       BEGIN Min := A[J];
             NMin := 1
       END
       ELSE  IF A[J] = Min THEN
             NMin := NMin + 1
    END;
    AVG := Sum/NUMBER;
    WRITELN;
    WRITELN('Maximum =',Max:4,', occurring',NMax:4, ' times');
    WRITELN;
    WRITELN('Minimum =', Min:4,', occurring', NMin:4,' times');
    WRITELN;
    WRITELN ('Average value (truncated) =', TRUNC(Avg):10);
    WRITELN ('Average value =', Avg : 20)
  END;
```

This procedure computes the minimum, maximum, and average values in array A. It also counts the number of times the minimum and maximum values occurred, and stores those numbers in NMin and NMax. The WRITELN statements print the results of each of those computations.

Min, Max, and Avg are formal variable parameters. Their values, as assigned in the procedure MIN_MAX_AVG, are returned to the calling program unit and can be used for further computations in the program. A is specified as a value parameter because its value does not change in the procedure.

The following is a valid procedure call to the procedure:

```
Min_Max_Avg(Minimum, Maximum, Average, ARR);
```

The values of the formal parameters Min, Max, and Avg are returned to the actual parameters Minimum, Maximum, and Average, which were defined in the main block of the program.

6.5 DECLARING A FUNCTION

A function is a group of statements that compute a scalar or pointer value. To declare a function, specify its heading and block in the procedure and function section.

Format 1

```
FUNCTION function id  ⟦(formal parameters)⟧ :result type;

     label section;
     constant section;
     type section;
     variable section;
     procedure-and-function section;

   BEGIN
     statement  ⟦;statement...⟧
   END;
```

Format 2

```
{⟦ [GLOBAL]  ⟧} FUNCTION function id  ⟦(formal parameters)⟧ ; ⟦FORWARD;⟧
{⟦ [FORTRAN] ⟧}
```

Format 3

```
FUNCTION function id  ⟦(formal parameters)⟧ ;{⟦ EXTERN ⟦AL⟧ ; ⟧}
                                              {⟦ FORTRAN;      ⟧}
```

Format 1 shows the format for a function that is included within the program that calls it.

Format 2 shows the format for a function that can be called externally, that is, it can be called from another program. Placing the FORTRAN declaration at the beginning of the line indicates that a FORTRAN program can call this procedure. The FORWARD declaration permits the use of forward references in the declarations section. Forward declarations are described in Section 6.6.

Format 3 shows the format for a function that is defined externally; the procedure is compiled separately from the source program that calls it. Placing the FORTRAN declaration at the end of the line indicates that the procedure being called by PASCAL is written in the FORTRAN language. Refer to Section 6.7.

function id

> specifies the identifier to be used as the name of the function.

formal parameters

> contain the names and types of the formal parameters. They optionally can include the reserved words VAR, PROCEDURE, and FUNCTION.

result type

> specifies the type of the function's result. The result must be a scalar or pointer value.

label section

declares local labels.

constant section

defines local constant identifiers.

type section

defines local types.

variable section

declares local variables.

procedure-and-function section

declares local procedures and functions.

statement

specifies an action to be performed by the function. A function can contain any of the statements described in Chapter 5. A function must contain a statement that assigns a value to the function identifier (for every potential path through the code). If it does not, the value of the function could be undefined.

A function consists of a heading and a block. The formal parameter list in the function heading is identical in format to the list in the procedure heading. The function block is similar in structure and contents to the main program, with the following exceptions:

- The function cannot contain a value initialization section.
- The function block ends with END followed by a semicolon (;), rather than a period (.) as in the main program. The function does not have a block if it is a forward declaration, or if it is defined externally (EXTERN(AL) or FORTRAN).

You must declare all variables that are local to the function, but you should not redeclare a variable, type, or constant with the same name as a formal parameter or the function identifier.

Each function must include a statement or statements that assigns a value of the result type to the function name. The last value that is assigned to the function name is returned to the calling program unit. To use the value, include a function call in the calling unit. Unlike a procedure call, a function call is not a statement. It simply represents a value of the function's result type.

Side Effects

A side effect is an assignment to a nonlocal variable, or to a VAR parameter, within a function block. Side effects can change the intended action of a program and therefore, should be avoided. The following program illustrates an example of a side effect.

```
PROGRAM Example (OUTPUT);
VAR X,Y : INTEGER;
    ANS1, ANS2 : BOOLEAN;

   FUNCTION Positive (ThisVar : INTEGER) : BOOLEAN;
   BEGIN
      Positive := FALSE;
      IF ThisVar > 0 THEN
      BEGIN
         X := ThisVar - 10;              (* Side effect on X *)
         Positive := TRUE
      END
   END;  (* end Positive*)

BEGIN                                    (* MAIN PROGRAM *)
   Y := 7;  X :=15;
   ANS1 := Positive(X) AND Positive(Y);
   WRITELN ('ANS1 equals',ANS1);
   Y := 7; X := 15;
   ANS2 := Positive(Y) AND Positive(X);
   WRITELN ('ANS2 equals',ANS2)
END.
```

This example generates the following output:

```
ANS1 equals          TRUE
ANS2 equals          FALSE
```

Thus, the output depends on which function call is evaluated first: Positive(Y) or Positive(X). PASCAL does not guarantee which part of an expression is evaluated first. The resulting value of a function should not depend on when the function is called, as it does in the example above. Therefore, you should avoid side effects on global variables.

Example 1

```
FUNCTION COUPONS :  REAL;
    VAR ANS : (YES, NO);
        AMOUNT, SUBT : REAL;
    BEGIN
       SUBT := 0;
       WRITELN ('Any coupons? Type yes or no.');
       READLN (ANS);
       IF ANS=YES THEN
           BEGIN
             WRITELN ('Type value of each coupon, one per line,
                      CTRL/Z when finished?');
             REPEAT
               READLN (AMOUNT);
               SUBT := SUBT + AMOUNT
             UNTIL EOF
           END;
       COUPONS := SUBT
END;                    (* End COUPONS* )
```

The function COUPONS computes the total value of a group of coupons. It uses only the three local variables, ANS, AMOUNT, and SUBT, and requires no parameters. The result of this function is the real total of the coupon values. The assignment statement, COUPONS := SUBT, assigns the result to the function identifier.

To use the function COUPONS, specify its name, as follows:

```
     TOTAL := SUBTOTAL - COUPONS;
```

The function call is treated as a real-valued expression in this statement. Note that you can use the function call in the same way that you can use a value of its result type.

Example 2

```
     FUNCTION SYMMETRY (VAR A : ARR) : BOOLEAN;
     (*This function returns true if the array A is symmetric; it
     returns false otherwise.*)

     VAR I, J : INTEGER;
     BEGIN
        SYMMETRY := TRUE;
        FOR I := 1 TO SIZE DO
           FOR J := I TO SIZE DO
              IF  A[I,J] <> A[J,I] THEN  SYMMETRY := FALSE
     END;    (* SYMMETRY *)
```

The function SYMMETRY uses one variable parameter, the array A. SYMMETRY returns a Boolean value; the result is TRUE if A is symmetric, and FALSE if A is not symmetric.

6.6 FORWARD DECLARATIONS

Normally, you must declare subprograms before you refer to them. However, a subprogram can reference another subprogram that has not yet been declared if you use a FORWARD declaration. The forward declaration provides the compiler with information about the forward-declared subprogram's formal parameters, and indicates that the block of the subprogram follows later in the source file.

For example, a complete declaration is impossible if two subprograms call each other recursively. Omitting the declaration is also impossible because PASCAL needs information about a subprogram's formal parameters before it can compile a reference to the subprogram. Therefore, you must forward-declare one of the recursive subprograms.

A forward declaration consists of the subprogram heading (including the formal parameter list, if any, and the result type, if it is a function) and the FORWARD directive without a subprogram block, for example:

```
     PROCEDURE CHESTNUT (BLD :REAL; DOC : CHAR; VAR ARC : REC);FORWARD;
```

This example declares the procedure CHESTNUT in a FORWARD declaration. The FORWARD declaration includes only the information shown in the example. It could also include FORTRAN or GLOBAL, as in Format 2 above.

When you specify the block of a forward-declared subprogram, you must not repeat the formal parameter list or the result type of a function. Except for these omissions, declare the heading and block in the normal way.

Example

```
     FUNCTION ADDER (OP1, OP2, OP3 : REAL) : REAL; FORWARD;

     PROCEDURE PRINTER (STUDENT : NAME_ARRAY) ;
         .
         .
         .
         BEGIN
         .
         .
         .
            Z := ADDER (A,B,C)
         END;

     FUNCTION ADDER (* OP1, OP2, OP3 : REAL : REAL*);
         .
         .
         .
         BEGIN
         .
         .
         .
            PRINTER ('Leonardo da Vinci');
         .
         .
         .
         END;
```

This example forward-declares the function ADDER. The block of the function appears after the declaration of the procedure PRINTER. Note that the heading of the ADDER block specifies its formal parameters and result type within comment delimiters. Although you must omit the parameter list and result type when you define the function block, inserting them as a comment is a good documentation practice.

6.7 EXTERNAL SUBPROGRAMS

The FORTRAN and EXTERNAL directives indicate procedures and functions that are defined external to a PASCAL program. With these directives, you can declare subprograms written in another language (such as FORTRAN or MACRO) and PASCAL subprograms that are compiled separately. In PASCAL, the FORTRAN directive should be used only for separately compiled routines written in FORTRAN or a language using the FORTRAN subprogram calling conventions. The EXTERNAL directive must be used only for separately compiled external routines written in PASCAL.

If you declare separately compiled PASCAL subprograms as EXTERNAL, their names must be unique within the first six characters. In addition, an external subprogram cannot have the same name as the main program.

Example 1

```
FUNCTION SCORE (RESULT : REAL) : REAL; EXTERNAL;
```

The function SCORE is a procedure in a library that exists on disk. This declaration declares SCORE as an external subprogram.

Example 2

```
PROCEDURE FORSTR(S : PACKED ARRAY[L..U:INTEGER] OF CHAR) FORTRAN;
```

This statement declares the FORTRAN procedure FORSTR. The formal parameter list specifies S as a conformant-array parameter.

6.8 MODULES FOR SEPARATE COMPILATION

By placing PASCAL procedures and functions in a MODULE, you can compile them separately from the main program. At load time, you specify the compiled files containing the main program and the modules to be loaded together in the executable image. The executable image can include any number of modules, and each module can contain any number of subprograms.

Format

```
⟦ [OVERLAID] ⟧ MODULE module name (program parameters);
label section;
constant section;
type section;
variable section;
procedure-and-function section;
END.
```

[OVERLAID]

Specifies that the module shares global values with the main program that calls it. If the module is OVERLAID; then the constant, type, and variable sections must be identical to those in the main program.

module name

specifies the identifier to be used as the name of the module.

program parameters

lists the external files. This list must be identical in order and in content to the list in the main program heading.

label section

declares global labels. PASCAL issues a warning-level message if a module contains a label section, but ignores the labels.

constant section

declares global constant identifiers as in a main program.

type section

defines global types as in a main program.

variable section

declares global variables as in a main program.

procedure-and-function section

declares the procedures and functions contained in the module.

A module is similar to a main program, except that it has no value initialization section and no executable section. Modules can contain the label, constant, type, variable, and procedure-and-function sections. (PASCAL issues a warning-level message if a module contains label declarations, but ignores the labels.) If the module is OVERLAID, then the constant, type, and variable sections and the program parameters must be identical to those in the main program. The procedure-and-function section defines the subprograms contained in the module.

To ensure that the program parameters and the constant, type, and variable sections are identical in the main program and in all modules, you can place them in a separate file. Then, you can use the %INCLUDE directive to insert the contents of the file into the main program and into all modules, instead of repeating all the declarations and definitions.

If a module shares global variables with a main program, both the module and the program headings must include the attribute [OVERLAID]. If the module heading does not contain [OVERLAID], then all its global variables are private to the module and cannot be accessed by the main program or other modules. Subprograms declared at the outermost level of a module can be declared and called from the main program (or from another module). You must declare the subprogram with the EXTERNAL modifier in the calling program unit and with the [GLOBAL] attribute in the module. Similarly, subprograms declared at the outermost level of the main program with the [GLOBAL] attribute can be declared as EXTERNAL in a module.

Each subprogram in the module can access data declared either locally to itself or by the main program.

Examples

```
[OVERLAID] MODULE SEP (INPUT, OUTPUT);
VAR I :  INTEGER;
PROCEDURE READER (N : INTEGER);
   VAR K,P : INTEGER;
   BEGIN
      I := 0;
      FOR K := 1 TO N DO
          BEGIN
              READ (P);
              IF P=0 THEN I := I + 1;
          END
   END;  (* READER *)
   END.  (*MODULE SEP *)
```

The MODULE SEP contains one procedure, READER. You can declare READER as an external subprogram in another module or in the main program. Because SEP contains the definition of a global data item, I, it is declared as an [OVERLAID] module. If you declare READER as an external subprogram, you must declare READER as [GLOBAL] in the module so that the main program can call it.

CHAPTER 7

INPUT AND OUTPUT

This chapter describes input and output (I/O) for PASCAL on TOPS-20. PASCAL provides predefined procedures to perform input and output to file variables. These procedures are divided into the following categories:

General Procedures

- OPEN -- associates a file with specified characteristics
- CLOSE -- closes a file
- FIND -- performs direct access to file components

Input Procedures

- RESET -- opens a file and prepares it for input
- GET -- reads a file component into the file buffer variable
- READ -- reads a file component into a specified variable
- READLN -- reads a line from a text file

Output Procedures

- REWRITE -- truncates a file to length zero and prepares it for output
- PUT -- writes the contents of the file buffer variable into the specified file
- WRITE -- writes specified values into a file
- WRITELN -- writes a line into a text file
- LINELIMIT -- terminates program execution after a specified number of lines have been written into a text file
- PAGE -- skips to the next page of a text file

In addition, you can use the predefined functions EOF and EOLN with text files.

The following sections describe:

- PASCAL file characteristics
- PASCAL record formats

- PASCAL input and output procedures
- Terminal I/O

The input and output procedures are presented in alphabetical order.

7.1 FILE CHARACTERISTICS

This section describes the organization of records and methods of accessing records.

The term file organization applies to the way records are physically arranged on a storage device. The term record access refers to the method used to read records from or write records to a file, regardless of the file's organization. A file's organization is specified when the file is created and cannot be changed. Record access is specified each time the file is opened and can vary.

7.1.1 File Names

The file name indicates the system name of a file that is represented by a PASCAL file variable in an OPEN procedure (Section 7.7). For the file name, you can specify a character-string expression that contains a TOPS-20 file specification or a logical name. Apostrophes are required to delimit a character-string constant or a logical name used as a file name.

7.1.2 Logical Names

The TOPS-20 operating system provides the logical name mechanism as a way of making programs device and file independent. If you use logical names, your PASCAL program need not specify the particular device on which a file resides or the particular file that contains data. Specific devices and files can be defined at run time.

A logical name is an alphanumeric string that you specify in place of a file specification. Logical names provide great flexibility because they can be associated not only with a complete file specification, but also with a device, a device and a directory, or even another logical name.

On TOPS-20 you can create a logical name and associate it with a file specification by means of the TOPS-20 DEFINE command. Thus, before program execution, you can associate the logical names in your program with the file specifications appropriate to your needs, for example:

```
    @DEFINE DATA:  PS:<BENJAMIN>TEST.DAT.2
```

This command creates the logical names DATA: and associates it with the file specification PS:<BENJAMIN>TEST.DAT.2. The system uses this file specification when it encounters the logical name DATA: during program execution, for example:

```
    OPEN (INDATA, 'DATA:', OLD);
```

In executing this PASCAL statement, the system uses the file specification PS:<BENJAMIN>TEST.DAT.2 for the logical name DATA:. To specify a different file when you execute the program again, issue another DEFINE command, for example:

```
    @DEFINE DATA: PS:<JENNIFER>REAL.DAT.7
```

This command associates the logical name DATA: with a different file specification and replaces the previous logical name assignment. The OPEN statement above now refers to the file PS:<JENNIFER>REAL.DAT.7. For more information about the use of logical names, refer to the TOPS-20 User's Guide.

7.1.3 File Organization

PASCAL supports sequential file organization. Sequential files consist of records arranged in the order in which they are written to the file. The first record written is the first record in the file; the second record written is the second record in the file, and so on. As a result, records can be added only at the end of the file.

7.1.4 Record Access

You specify record access mode as a parameter to the OPEN procedure. PASCAL provides two ways of accessing records:

- Sequential
- Direct

If you select sequential access mode, records are written to or read from the file, starting at the beginning and continuing through the file one record after another.

Having sequential access to a file means that you can read a particular record only after reading all the records preceding it. New records can be written only at the end of a file that is open for sequential access.

If you select direct access mode, you can specify the order in which records are accessed. Each FIND procedure call must include the relative record number indicating the record to be read. You can directly access a file only if it contains fixed-length records, resides on disk, and is open for input (reading).

7.2 RECORD FORMATS

Records are stored in one of two formats:

- Fixed length
- Variable length

You can access fixed-length records in either sequential or direct mode. Variable-length records can be accessed only in sequential mode.

7.2.1 Fixed-Length Records

When you specify fixed-length records, you are specifying that all records in the file contain the same number of bytes. A file opened for direct access must contain fixed-length records to allow the record location to be computed correctly. All binary files (that is, all files except TEXT files) must have fixed-length records. PASCAL does not support binary files with variable-length records.

7.2.2 Variable-Length Records

Variable-length records can contain any number of bytes up to the buffer size specified when the file was opened. TEXT files must have variable-length records. PASCAL does not support TEXT files with fixed-length records.

7.3 THE CLOSE PROCEDURE

The CLOSE procedure closes an open file.

Format

```
    CLOSE (file variable);
```

where:

file variable	specifies the file to be closed.

Execution of this procedure causes the system to close the file. Each file is automatically closed upon exit from the procedure in which it is declared, except those which have been dynamically allocated with the procedure NEW. These files should be explicitly closed; if not, they are automatically closed when the program ends, or when they are DISPOSED.

You can close only files that have been opened explicitly with the OPEN procedure or implicitly by the RESET or REWRITE procedure. Therefore, you cannot close the predeclared file variables INPUT and OUTPUT.

Example

```
    CLOSE (Albums);
```

This procedure closes the file Albums to further access, and deletes the file if it is internal to the current program.

7.4 THE FIND PROCEDURE

The FIND procedure positions a file pointer at a specified component in the file.

Format

```
    FIND (file variable, integer expression);
```

where:

file variable	specifies a file that is open for direct access. The file must have fixed-length records.
integer expression	specifies the positive integer expression indicating the component at which to position the file. The component number must not be less than or equal to zero.

The FIND procedure allows you to directly access the components of a file. You can use the FIND procedure to move forward or backward in a file. The file must be open for direct access. That is, you must have specified DIRECT in the OPEN statement for that file. In addition, the file must have fixed-length records.

After execution of the FIND procedure, the file is positioned at the specified component. The file buffer variable assumes the value of the component, for example:

```
FIND (Albums, 4);
```

As the result of this statement, the file position moves to the fourth component in the file Albums. The file buffer variable Albums^ assumes the value of the fourth component.

If you specify a component beyond the end of the file, no error occurs.

You can use the FIND procedure only when reading a file. If the file is open for output (that is, with REWRITE), a call to FIND results in a run-time error.

Example 1

```
BEGIN
   FIND (Albums, Current + 2);
   .
   .
   .
END;
```

If the value of Current is 6, this statement causes the file position to move to the eighth component. The file buffer variable Albums^ assumes the value of the component.

Example 2

```
BEGIN
   FIND (Albums, Current-1);
   .
   .
   .
END;
```

If the value of Current is 6, this statement causes the file position to move backward one component to the fifth component.

7.5 THE GET PROCEDURE

The GET procedure reads the next component of a file into the file buffer variable.

Format

```
     GET (file variable);
```

where:

file variable	specifies the file to be read.

Before you use the GET procedure to read one or more file components, you must have called the RESET procedure to prepare the file for reading (input). RESET moves the file position to the first component and assigns its value to the file buffer variable.

As a result of the GET procedure, the file position moves to the next component of the file. The file buffer variable takes on the value of that component, for example:

```
     RESET (Books);
     Newrec := Books^;
     GET (Books);
```

After execution of the RESET procedure, the file buffer variable Books^ is set to the first component of the file. The assignment statement assigns this value to the variable Newrec. The GET procedure then assigns the value of the second component to Books^, moving the file position to the second component. The next GET procedure moves the file position to the third component, as shown in Figure 7-1.

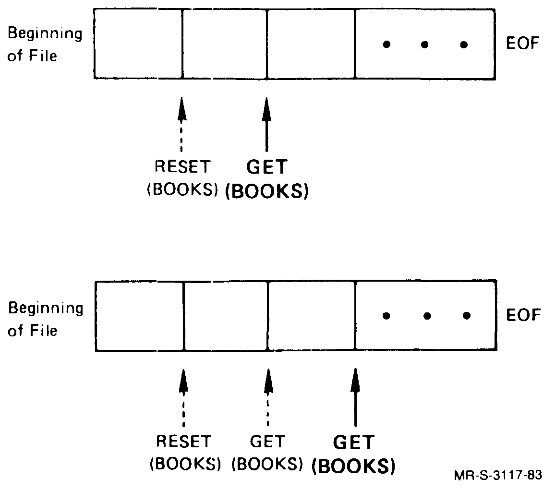


Figure 7-1: File Position After GET

By repeatedly using GET statements, you can read sequentially through the file.

When you reach the end of the file and you request a GET operation, EOF automatically becomes TRUE, and the file buffer variable becomes undefined. When EOF is TRUE, you cannot use the GET procedure. PASCAL signals a run-time error, and program execution is aborted.

Example

```
BEGIN
   GET (Phones);
   .
   .
   .
END;
```

This example reads the next component of the file Phones into the file buffer variable Phones^. EOF(Phones) must be FALSE; if it is TRUE, an error occurs.

7.6 THE LINELIMIT PROCEDURE

The LINELIMIT procedure terminates execution of the program after a specified number of lines have been written into a text file.

Format

```
LINELIMIT (file variable, n);
```

where:

file variable	specifies the text file to which this limit applies.
n	represents a positive integer expression indicating the number of lines that can be written to the file before execution terminates.

When PASCAL initializes a text file, it specifies a large default line limit. You can override this limit by calling LINELIMIT with your desired value.

After the number of lines output to the file has reached the line limit, program execution terminates.

Example

```
BEGIN
   LINELIMIT (Debts,100);
   .
   .
   .
END;
```

Execution of the program terminates after 100 lines have been written into the text file Debts.

7.7 THE OPEN PROCEDURE

The OPEN procedure does not actually open a file but rather allows you to specify file attributes. You cannot use OPEN on a file that has had a RESET or REWRITE done, or on the predeclared file INPUT.

Format 1

```
     OPEN (file variable  ⟦ ,file name ⟧   ⟦ ,history ⟧
          ⟦ ,record length ⟧   ⟦ ,record-access-method ⟧
          ⟦ ,record type ⟧   ⟦  ,carriage control ⟧  );
```

Format 2

```
     OPEN (FILE_VARIABLE := file variable
           ⟦ ,FILE_NAME := file name ⟧
           ⟦ ,HISTORY := history status ⟧
           ⟦ ,RECORD_LENGTH := positive integer ⟧
           ⟦ ,RECORD_ACCESS_METHOD := record-access-mode ⟧
           ⟦ ,RECORD_TYPE := record type ⟧
           ⟦ CARRIAGE_CONTROL := carriage control ⟧   );
```

where:

file variable	specifies the PASCAL file variable associated with the file. You cannot open the predeclared file variable INPUT. (internal files) This parameter is ignored for internal files. The system creates a unique name for each internal file. (external files)
file name	provides information about the file to TOPS-20. The file name can be a variable or constant identifier defined as type PACKED ARRAY [1..n] OF CHAR, or a file specification enclosed in apostrophes (for example, 'PS:<MASELLA>BOOKS.DAT'). If you omit the file name, PASCAL will first attempt to use the file variable identifier as a logical name. If that name is not defined, PASCAL will use the defaults shown in Table 7-1.

The file variable and the file name parameters designate the file to be opened. The remaining parameters specify attributes for the file and are summarized in Table 7-2, in Section 7.7.5. Except for the file variable name, all parameters are optional. Any parameters you specify, however, must be in the order shown above unless you use keyword syntax.

You can specify either the value of the parameter or the keyword and the value of the parameter. You can also use a combination.

To specify only the value without the keyword, place each parameter in the same order shown in the format. If a particular parameter is not used, then a comma may be inserted. PASCAL generates a warning message if the position of an unused parameter is not indicated by a comma. However, the correct default for the missing parameter is used in either case.

To use a keyword, specify the keyword and its associated value. When you use a keyword, you have to specify only the parameters that are used; it is not necessary to insert a comma to indicate unused parameters. Keyword parameters can be placed in any order; they do not have to be in the same order as shown in the format.

You can also use a combination of values and keywords with values. However, once you use a keyword within the statement, subsequent values must be associated with a keyword.

Table 7-1: Default Values for TOPS-20 External File Specifications

Element	**Default**
Device	Current user device
Directory	Current user directory
File name	PASCAL file variable name, truncated to first 39 characters
File type	DAT
Generation number	OLD: highest current number NEW: highest current number + 1

Because the RESET and REWRITE procedures actually open files, you need not always use the OPEN procedure. RESET and REWRITE impose the defaults for the TOPS-20 file specification, file status, record length, record access mode, record type, and carriage control shown in Table 7-1 and Table 7-2. For the file status attribute, RESET uses a default of OLD, and REWRITE uses a default of NEW. You must use the OPEN procedure for the following:

- To open a file for DIRECT access by the FIND procedure
- To specify a buffer size other than 133 for a text file
- To open any file with other than the default file name

7.7.1 File History -- NEW, OLD, READONLY, or UNKNOWN

The file status indicates whether the specified file exists or must be created. The possible values are:

NEW
OLD
READONLY
UNKNOWN

A file status of NEW indicates that a new file must be created with the specified attributes. NEW is the default value.

If you specify OLD, the system tries to open an existing file. An error occurs if the file cannot be found. OLD is invalid for internal files, which are newly created each time the declaring program or subprogram is executed.

A file status of READONLY indicates that an existing file is being opened only for reading. An error occurs if you try to write to a file that has been opened with READONLY.

If you specify UNKNOWN, the system first tries to open an existing file. If an existing file cannot be found, a new file is created with the specified attributes.

If you specify READONLY, the system generates an error if a REWRITE is performed on the file. READONLY implies OLD.

7.7.2 Record Length

The record length parameter specifies the record length used in the file. Any positive integer can be used.

7.7.3 Record Access Mode -- SEQUENTIAL or DIRECT

The record access mode specifies the access to the components of the file. The modes are:

DIRECT
SEQUENTIAL

In SEQUENTIAL mode, you can access files with fixed- or variable-length records. The default access mode is SEQUENTIAL.

DIRECT mode allows you to use the FIND procedure to access files with fixed-length records. You cannot access a file with variable-length records in DIRECT mode.

7.7.4 Record Type -- FIXED or VARIABLE

The record type specifies the structure of the records in a file. The record types are:

FIXED
VARIABLE

A value of FIXED indicates that all records in the file have the same length. A value of VARIABLE indicates that the records in the file can vary in length. FIXED is the default for non-TEXT files; VARIABLE is the default for TEXT files.

7.7.5 Carriage Control -- LIST, CARRIAGE, or NOCARRIAGE

The carriage control option specifies the carriage control format for a text file. The options are:

LIST
CARRIAGE
FORTRAN
NOCARRIAGE
NONE

A value of LIST indicates single spacing between components. LIST is the default for all text files, including the predefined file OUTPUT.

The CARRIAGE option indicates that the first character of every output line is a carriage control character. These characters and their effects are summarized in Table 7-4.

FORTRAN is equivalent to CARRIAGE.

NOCARRIAGE means that no carriage control applies to the file. In particular, WRITELN will not output an EOLN to a NOCARRIAGE file, and the PAGE procedure will cause a run-time error.

NONE is equivalent to NOCARRIAGE.

Table 7-2 summarizes the file attributes.

Table 7-2: Summary of File Attributes

Parameter	Possible Values	Default
File status	OLD, NEW, READONLY or UNKNOWN	NEW
Record length	any positive integer	133
Record-access mode	DIRECT or SEQUENTIAL	SEQUENTIAL
Record type	FIXED or VARIABLE	VARIABLE for text files; FIXED for non-text files.
Carriage control	LIST, CARRIAGE, FORTRAN, NOCARRIAGE, or NONE	LIST for all text files; NOCARRIAGE for all other files

7.7.6 Examples

Example 1

```
VAR Userguide : TEXT;

BEGIN
   OPEN (Userguide);
   .
   .
   .
END;
```

In this example, the OPEN procedure specifies only the file variable so no defaults for the file will be changed. This usage of OPEN essentially causes no action.

Example 2

```
BEGIN
   OPEN (Userguide,,,80);
   .
   .
   .
END;
```

The OPEN statement sets the record length for USERGUIDE to 80 characters.

Example 3

```
BEGIN
   OPEN (OUTPUT,,,,,CARRIAGE);
   .
   .
   .
END;
```

This example causes the system to interpret the first character of each line written to the predeclared file OUTPUT as a carriage control character. When you call OPEN for the predeclared file OUTPUT, you can specify only a carriage control option. If you include any other parameters, an error occurs.

Example 4

```
BEGIN
   OPEN (Albums, 'PS:<JENNIFER>INVENT', OLD, ,DIRECT);
   .
   .
   .
END;
```

The file variable albums will be associated with the file specification PS:<JENNIFER>INVENT. A RESET will initiate reading of the existing (OLD) file, or cause an error if the file does not exist. The file will be opened for direct access; that is you can use the FIND procedure with this file. A REWRITE will ignore the OLD parameter.

Example 5

```
BEGIN
   OPEN (Solar, 'Energy:', NEW, , , FIXED);
   .
   .
   .
END;
```

Assuming that Energy is defined as a logical name, this statement causes a RESET or REWRITE to create a file with the specification designated by the logical name Energy. The identifier Solar is used within the program to refer to the TOPS-20 logical name. The file is created with fixed-length records. Default values are used for the record length and the record access parameters.

Example 6

```
BEGIN
   OPEN (File_Name := 'PS:<SMITH>PLAN.DAT',RECORD_TYPE := VARIABLE,
         File_Variable:=Plans);
   .
   .
   .
END;
```

The file variable plans is associated with the file PLAN.DAT on PS: with a directory of <SMITH>. The file name and record type parameters use keywords. Because the file name keyword is used, each subsequent parameter must use a keyword. It is not necessary to indicate each unused parameter with a comma when keywords are used.

7.8 THE PAGE PROCEDURE

The PAGE procedure skips to the next page of a text file.

Format

```
    PAGE (file variable);
```

where:

file variable   specifies a text file.

Execution of the PAGE procedure causes the system to flush the contents of the record buffer, then skip to the next page of the specified text file. The next line written to the file begins on the first line of a new page. You can use this procedure only on text files. If you specify a file of any other type, PASCAL issues an error message.

The value of the page eject record that is output to the file depends on the carriage control format for that file. When CARRIAGE is enabled, the page eject record is equivalent to the carriage control character '1'. When LIST is enabled, the page eject record is a form feed character. When NOCARRIAGE is enabled, the PAGE procedure generates an error.

Example 1

```
    BEGIN
       PAGE (Userguide);
       .
       .
       .
    END;
```

This example causes a page eject record to be written in the text file Userguide.

Example 2

```
    BEGIN
       PAGE (OUTPUT);
       .
       .
       .
    END;
```

This example calls the PAGE procedure for the predeclared file OUTPUT. As a result of this procedure, a page eject record is output at the terminal (in interactive mode) or in the batch log file (in batch mode).

7.9 THE PUT PROCEDURE

The PUT procedure appends a new component to the end of a file.

Format

```
     PUT (file variable);
```

where:

file variable	specifies the file to which one or more components will be written.

Before executing the PUT procedure, you must have executed the REWRITE procedure. REWRITE clears the file and sets EOF to TRUE, preparing the file for output. If EOF is FALSE, the PUT procedure fails; a run-time error occurs; and program execution is terminated.

The PUT procedure writes the value of the file buffer variable at the end of the specified file. After execution of the PUT procedure, the value of the file buffer variable becomes undefined. EOF remains TRUE.

Example

```
PROGRAM Bookfile (INPUT,OUTPUT,Books);

TYPE String = PACKED ARRAY[1..40] OF CHAR;
     Bookrec = RECORD
                 Author : String;
                 Title : String
                 END;
VAR  Newbook : Bookrec;
     Books : FILE OF Bookrec;
     N : INTEGER;

BEGIN
     REWRITE (Books);
     FOR N := 1 TO 10 DO BEGIN
        WITH Newbook DO BEGIN
             WRITE ('Title:');
             READ (Title);
             WRITE ('Author:');
             READ (Author);
             END;
     Books^ := Newbook;
     PUT (Books)
     END
END.
```

This program writes the first 10 records into the file Books. The records are input from the terminal to the record variable Newbook. They consist of two 40-character strings denoting a book's author and title. The FOR loop accepts 10 values for Newbook, assigning each new record to the file buffer variable Books^. The PUT statement writes the value of Books^ into the file for each of the 10 records input.

7.10 THE READ PROCEDURE

The READ procedure reads one or more file components into a variable of the component type.

Format

```
READ ( ⟦file variable,⟧ variable name ⟦,variable name... ⟧ );
```

where:

file variable	specifies the input file. If you omit the file variable, PASCAL uses INPUT by default.
variable name	specifies the variable into which the file component(s) are read. For a text file, many file components can be read into one variable.

By definition, the READ procedure for a nontext file performs an assignment statement and a GET procedure for each variable name. Thus, the procedure call

```
READ (file variable, variable name);
```

is equivalent to

```
variable name := file variable^;
GET (file variable);
```

The READ procedure reads from the file until it has found a value for each variable in the list. The first value read is assigned to the first variable in the list; the second value is assigned to the second variable, and so on. The values and the variables must be of compatible types.

For a text file, more than one file component (that is, more than one character) can be read into a single variable. For example, many text file components can be read into a string or numeric variable. The READ procedure repeats the assignment and GET process until it has read a sequence of characters that represent a value for the next variable in the parameter list. It continues to read components from the file until it has assigned a value to each variable in the list.

Values from a text file can be read into variables of integer, real, character, string, and enumerated types. In the file, values to be read into integer and real variables must be separated by spaces or must be put on new lines. Values to be read into character variables, however, must not be separated because they are read literally, character-by-character. Constants of enumerated types must be separated by at least one space. Any other character that is invalid in an identifier terminates the constant. Only the first 31 characters of the constant are significant; PASCAL ignores any remaining characters.

You can use READ to read a sequence of characters from a text file into a string (that is,a variable of type PACKED ARRAY[1..n] OF CHAR). PASCAL assigns successive characters from the file to elements of the array, in order, until each element has been assigned a value. If any characters remain on the line after the array is full, the next READ begins with the next character on that line. If the end of the line is encountered before the array is full, the remaining elements are assigned spaces.

READ does not read past EOLN if it is reading into a string type. Instead, READ continues to return blanks until the EOLN is explicitly passed by using READLN.

If you call READ when the file is positioned at the end of a line, the file position moves to the beginning of the next line, unless it is a string variable. Characters are then read into the specified starting variable. If this line is empty, the string is filled with spaces.

Every text file ends with an end-of-line mark and an end-of-file mark. Therefore, the function EOF never becomes TRUE when you are reading strings with the READ procedure. To test EOF when reading strings, use the READLN procedure.

Example 1

```
BEGIN
   READ (Temp, Age, Weight);
   .
   .
   .
END;
```

Assume that Temp, Age, and Weight are real variables, and you type in the following values:

```
98.6 11.0 75.0
```

The variable Temp takes on the value 98.6; Age takes on the value 11.0; and Weight takes on the value 75.0. Note that you need not type all three values on the same line.

Example 2

```
TYPE String = PACKED ARRAY [1..20] OF CHAR;
VAR Names : TEXT;
     Pres, Veep :  String;

BEGIN
   READ(Names, Pres, Veep);
   .
   .
   .
END;
```

This program fragment declares and reads the file Names, which contains the following characters:

```
John F. Kennedy     Lyndon B. Johnson   Lyndon B. Johnson    <EOLN>
Hubert H. Humphrey  <EOLN>
Richard M. Nixon    Spiro T. Agnew      <EOLN>
```

The first call to the READ procedure sets Pres equal to the 20-character string 'John F. Kennedy   ' and Veep equal to 'Lyndon B. Johnson   '. The second call to the procedure assigns 'Lyndon B. Johnson   ' to Pres and spaces to Veep. Unless READLN is used to read past the EOLN, READ continues to assign spaces.

7.11 THE READLN PROCEDURE

The READLN procedure reads lines of data from a text file.

Format

```
READLN ⟦( ⟦file variable,⟧ variable name ⟦,variable name...⟧ )⟧ ;
```

where:

file variable	specifies the name of the text file to be read. If you do not specify a file variable, PASCAL uses INPUT by default.
variable name	specifies the variable into which a value will be read. If you do not specify any variable names, READLN skips a line in the specified file.

The READLN procedure reads values from a text file. After reading values for all the listed variables, the READLN procedure skips over any characters remaining on the current line and positions the file at the beginning of the next line. All the values need not be on a single line; READLN continues until values have been assigned to all the specified variables, even if this process results in the reading of several lines of the input file. READLN performs the following sequence:

```
READ (file variable, variable name...);
READLN (file variable);
```

EOLN(file variable) is TRUE only if the new line is empty.

You can use the READLN procedure to read integers, real numbers, characters, strings, and constants of enumerated types. The values in the file must be separated as they are for the READ procedure.

The READLN statement automatically pads strings. Thus, it is not necessary to pad strings with spaces to match the variable size if you are using the predefined file INPUT or reading from a file defined as TEXT.

If EOLN() is TRUE when you call READLN, the first value read is the first value in the next line, unless you are reading a character. If you are reading a character, the first value read is a space.

Example

```
TYPE String = PACKED ARRAY [1..20] OF CHAR;
VAR Names : TEXT;
    Pres, Veep : String;
    .
    .
    .
    WHILE NOT EOF (Names) DO;
      BEGIN
      READLN (Names, Pres, Veep);
      .
      .
      .
      END;
```

This program fragment declares and reads the file Names, which contains the following characters:

```
     John F. Kennedy     Lyndon B. Johnson   Lyndon B. Johnson   <EOLN>
     Hubert H. Humphrey  <EOLN>
     Richard M. Nixon    Spiro T. Agnew      <EOLN>
     <EOLN>
     <EOF>
```

The READLN procedure reads the values 'John F. Kennedy ' for Pres and 'Lyndon B. Johnson ' for Veep. It then skips to the next line, ignoring the remaining characters on the first line. Subsequent execution of the procedure assigns the value 'Hubert H. Humphrey ' to Pres and sets Veep to all blanks, because READ of a string will not go past EOLN. The next execution of the procedure assigns the value 'Richard M. Nixon ' to Pres and 'Spiro T. Agnew ' to Veep, then skips to the next line. The last execution of READLN sets both Pres and Veep to all blanks, and skips the EOLN, which causes EOF to become TRUE, so the loop exits.

7.12 THE RESET PROCEDURE

The RESET procedure readies a file for reading by setting the file pointer to the first component in the input file.

Format

```
     RESET (file variable);
```

where:

file variable	specifies the file to be read.

If the file is not already open, RESET opens it using the defaults listed in Table 7-1 and Table 7-2. To open a file that does not use default values, use the OPEN statement.

After execution of RESET, the file is positioned at the first component; and the file buffer variable contains the value of this component. The arrow in Figure 7-2 shows the file position after RESET. If the file is empty, EOF is TRUE; otherwise, EOF is FALSE. If the file does not exist, RESET returns an error at run time; RESET does not create the file.

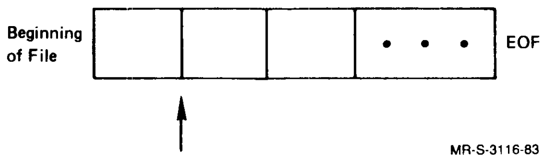


Figure 7-2: File Position after RESET

You must call RESET before reading any file except the predeclared file INPUT. If you call RESET for the predeclared file INPUT or OUTPUT, a run-time error occurs.

Examples

Example 1

```
BEGIN
   OPEN (Phones,'Phones.Dat',,DIRECT);
   RESET (Phones);
   .
   .
   .
END;
```

These statements open the file variable Phones for direct access on input. After execution of the OPEN and RESET procedures, you can use the FIND procedure for direct access to the components of the file Phones.

Example 2

```
BEGIN
   RESET (Weights);
   .
   .
   .
END;
```

If the file variable Weights is already open, this statement enables reading and sets Weights^ to the first file component. If the file is not open, this statement causes the system to search for the file designated by the logical name Weights: If no such logical name is assigned, the system searches for the file WEIGHTS.DAT on the user's default device and directory. If the file exists it is opened for sequential read access. If the file does not exist, a run-time error occurs.

7.13 THE REWRITE PROCEDURE

The REWRITE procedure readies a file for output by setting the file pointer to the first component of the output file.

Format

```
REWRITE (file variable);
```

where:

file variable	specifies the file to be enabled for output.

If the file does not exist, REWRITE creates and opens it using the defaults listed in Table 7-1 and Table 7-2. If the file exists, REWRITE supersedes it using the defaults listed in Table 7-1 and Table 7-2. To open a file that does not use default values, use the OPEN statement.

You must call REWRITE before writing any file except the predeclared file OUTPUT. If you call REWRITE for the predeclared file INPUT or OUTPUT, a run-time error occurs.

The REWRITE procedure sets the file to length zero and sets EOF to TRUE. You can then write new components into the file with the PUT, WRITE, or WRITELN procedure (WRITELN is defined only for text files). After the file is open, successive calls to REWRITE close and supersede the existing file; that is, they create new versions of the file.

To update an existing file, you must copy its contents to another file, specifying new values for the components that you need to update.

Example 1

```
     BEGIN
        REWRITE (Storms);
        .
        .
        .
     END;
```

If the file variable Storms is already open, this statement enables writing and sets the file position to the beginning of the file. If Storms is not open, a new version is created with the same defaults as for the OPEN procedure.

Example 2

```
     BEGIN
        OPEN (Results, 'PS:<CHEN>ISSUES.DAT',OLD,,FIXED);
        REWRITE (Results);
     END;
```

The OPEN procedure sets defaults for the file variable Results, which is associated with the file ISSUES.DAT in directory PS:<CHEN>. The REWRITE procedure discards the current contents of the file Results and sets the file position at the beginning of the file. After execution of this statement, EOF(Results) is TRUE.

7.14 THE WRITE PROCEDURE

The WRITE procedure writes data into a file.

Format

```
     WRITE ( [[ file variable, ]] print list );
```

where:

file variable	specifies the file to be written. If you omit the file variable, PASCAL uses OUTPUT by default.
print list	specifies the values to be output, separated by commas. The print list can contain constants, variables, and expressions. For nontext files, the items in the print list must be compatible with the file component type.

By definition, the WRITE procedure for a nontext file performs an assignment statement and a PUT procedure for each variable name. Thus, the following procedure calls are equivalent:

1. WRITE (file variable, variable name);

2. file variable^ := variable name;
   PUT (file variable)

For text files, the WRITE procedure converts each item in the print list to a sequence of characters. The WRITE procedure repeats the assignment and PUT process until all the items in the list have been written in the file.

The print list can specify constants, variable names, array elements, and record fields, with values of any scalar type. Each value is output with a minimum field width, as specified in Table 7-3.

Table 7-3: Default Values for Field Width

Type of Variable	**Number of Characters Printed**
Integer	12
Real	16
Double	24
Boolean	16
Character	1
Enumerated	31
String	Length of string

You can override these defaults for a particular value by specifying a field width in the print list. The field width specifies the minimum number of characters to be output for the value. The following is the format of the field-width specification:

```
variable name : minimum : fraction
```

Both minimum and fraction represent positive integer expressions. The minimum indicates the minimum number of characters to be output for the value. The fraction, which is permitted only for real numbers, indicates the number of digits to the right of the decimal point.

The following rules apply to designating field-width parameters in output procedures:

1. If a real value does not have the function parameter, PASCAL prints the value in floating-point format.

2. If the print field is wider than necessary, PASCAL prints the value with the appropriate number of leading blanks.

3. If the print field is too narrow, PASCAL treats the different kinds of write parameters as follows:

   - Strings and nonnumeric scalar values are truncated on the right to the specified field width.

   - Integers and real numbers in decimal format are printed using the full number of characters needed for the value, thus overriding the field-width specification.

   - Real and double values in floating-point format are printed in a field of at least eight characters (for example, -1.0E+00). All real values in either format are printed with a leading blank if they are positive and a leading minus sign if they are negative.

By default, PASCAL prints real numbers in floating-point format. Each real number is preceded by at least one blank, for example:

```
WRITE (Shoesize);
```

If the value of Shoesize is 12.5, this statement produces the following output:

```
1.25000000E+01
```

To print the value in decimal format, you must specify a field width as in this example:

```
WRITE (Shoesize:5:1);
```

The first integer indicates that a minimum of five characters will be output. The minimum includes the leading blank, the sign (if any), and the decimal point. The second integer specifies one digit to the right of the decimal point. This statement results in the following output:

```
12.5
```

If the print field is wider than necessary, PASCAL prints the value with leading blanks.

If you try to print a nonnumeric value in a field that is too narrow, PASCAL truncates the value on the right to fit into the field. For integers, however, it prints the entire value without truncation. PASCAL widens the field to eight characters for real and double-precision numbers in exponential notation. It does not truncate real and double-precision numbers in decimal notation.

For a variable of an enumerated type, PASCAL prints the constant identifier denoting the variable's value. Because PASCAL ignores any characters beyond the thirty-first in an identifier, only the first 31 characters of a long identifier appear, for example:

```
VAR Color : (Blue,Yellow,Black,Slightly_Pale_Peach_Summer_Sunset);
BEGIN
   WRITE ('My favorite color is ',Color:35);
     .
     .
     .
END;
```

When the value of Color is Yellow, the following is printed:

```
My favorite color is              YELLOW
```

When the value of Color is Slightly_Pale_Peach_Summer_Sunset, however, the following appears:

```
My favorite color is    SLIGHTLY_PALE_PEACH_SUMMER_SUNS
```

Although the field width specified is wide enough for all 33 characters in the identifier, PASCAL ignores the last two characters and prints two leading blanks. Note that constants of enumerated types are printed in all uppercase characters.

If you open the predeclared file OUTPUT with the carriage control option LIST, PASCAL allows you to use the WRITE procedure to prompt for input at the terminal. Each time you read from INPUT, the system checks for any output in the terminal record buffer. If the buffer contains any characters, the system prints them at the terminal, but suppresses the carriage return at the end of the line. The output text appears as a prompt, and you can type your input on the same line, for example:

```
WRITE ('Name three presidents:');
READ (Pres1, Pres2, Pres3);
```

When PASCAL executes the READ procedure, it finds the output string waiting to be printed. PASCAL prints the prompt at the terminal, leaving the carriage just after the colon (:). You can then begin typing input on the same line as the prompt.

Prompting works only for the predeclared files INPUT and OUTPUT. For any other files, no output is written until you fill the record buffer or start a new line.

Example 1

```
TYPE String = PACKED ARRAY [1..20] OF CHAR;
VAR Names :  FILE OF String;
    Pres :  String;
BEGIN
   WRITE (Names, 'Millard Fillmore    ', Pres);
     .
     .
     .
END;
```

This example writes two components in the file Names. The first is the 20-character string constant 'Millard Fillmore '. The second is the string variable Pres.

Example 2

```
    BEGIN
       WRITE (Num1:5:1,' and',Num2:5:1,' sum to',(Num1+Num2):6:1);
       .
       .
       .
    END;
```

If you specify an expression, PASCAL prints its value. For example, if Num1 equals 71.1 and Num2 equals 29.9, this statement prints:

```
    71.1 and 29.9 sum to 101.0
```

Note that each of the real numbers is preceded by a space.

Example 3

```
    VAR Rainamts : FILE OF REAL;
        Avgrain,Maxrain,Minrain : REAL;
    BEGIN
       WRITE (Rainamts,Avgrain,Minrain,0.312,Maxrain);
       .
       .
       .
    END;
```

The file Rainamts contains real numbers indicating amounts of rainfall. The WRITE procedure writes the values of the variables Avgrain and Minrain into the file, followed by the real constant 0.312 and the value of the variable Maxrain.

7.15 THE WRITELN PROCEDURE

The WRITELN procedure writes a line of data in a text file.

Format

```
    WRITELN ( [[ file variable, ]] print list);
```

where:

file variable	specifies the text file to be written. If you omit the file variable, PASCAL uses OUTPUT by default.
print list	specifies the values to be output, separated by commas. The print list can specify constants, variable names, array elements, and record fields, with values of any scalar type. Output of strings is also permitted. Each value is output with a minimum field width.

The WRITELN procedure writes the specified values into the text file, then starts a new line, for example:

```
     WRITELN (Userguide, 'This manual describes how you interact');
```

As a result of this statement, the system writes the string in the text file Userguide and skips to the next line.

When you open a text file, you can specify the CARRIAGE option for carriage-control format. If you select CARRIAGE format, the first character of each output line is treated as a carriage-control character when output is directed to carriage-control devices such as the terminal and the line printer. If output is not directed to a carriage-control device, the carriage-control character is written into the file and will be read when you open the file for input. Table 7-4 summarizes the carriage-control characters and their effects.

For carriage-control purposes, any characters other than those listed in the table are ignored.

The carriage-control character must be the first item in the WRITELN print list. For example, if the text file Tree is open with the CARRIAGE option, you can use the following statement:

```
     WRITELN (Tree,' ',String1,String2);
```

The first item in the print list is a space character. The space indicates that the values of String1 and String2 are printed beginning on a new line when the file is output to a terminal, line printer, or similar carriage-control device.

Table 7-4: Carriage-Control Characters

Character	**Meaning**
'+'	Overprinting: starts output at the beginning of the current line
space	Single spacing: starts output at the beginning of the next line
'0'	Double spacing: skips a line before starting output
1'	Paging: starts output at the top of a new page

If you specify CARRIAGE but use an invalid carriage-control character, the first character in the line is ignored. The output appears with the first character truncated.

Example 1

```
     BEGIN
        WRITELN (Class[1]:2,' is the grade for this student.');
        .
        .
        .
     END;
```

This example writes an element of the character array Class to the file OUTPUT. The value is written with a minimum field width of 2.

Example 2

```
     BEGIN
        WRITELN;
        .
        .
        .
     END;
```

If you specify WRITELN without a file variable or print list, PASCAL ends the printing of the current line on the standard output device (usually the terminal).

Example 3

```
     TYPE String : PACKED ARRAY [1..40] OF CHAR;
     VAR  Newhires : TEXT;
          N : INTEGER:
          Newrec : RECORD
                   ID : INTEGER;
                   Name : String;
                   Address : String;
                   Salary : String
                   END;
     BEGIN
        OPEN (Newhires, CARRIAGE);
        WITH Newrec DO BEGIN
             WRITELN (Newhires, '1New hire #',ID,'IS ',Name);
             WRITELN (Newhires, ' ', Name, '# Lives at:');
             WRITELN (Newhires);
             WRITELN (Newhires, ' ', Address)
             END;
        .
        .
        .
     END;
```

This example writes four lines in the text file Newhires. The output starts at the top of a new page, and fits the following format:

```
     New hire # 73 is Irving Washington
     Irving Washington lives at:

     22 Chestnut St, Seattle, Wash.
```

7.16 TERMINAL I/O

The PASCAL language requires that the file buffer always contain the next file component that will be processed by the program. This requirement can cause problems when the input to the program depends on the output most recently generated. To alleviate such problems in the processing of the predeclared text files INPUT and OUTPUT, PASCAL uses a technique called delayed device access, also know as lazy lookahead.

As a result of delayed device access, an item of data is not retrieved from a physical file device and inserted in the file buffer until the program is ready to process it. The file buffer is filled when the program makes the next reference to the file. A reference to the file consists of any use of the file buffer variable, including its implicit use in the GET, READ, and READLN procedures, or any test for the status of the file, namely, the EOF and EOLN functions.

The RESET procedure initiates the process of delayed device access. RESET is done automatically on the predeclared file INPUT. RESET expects to fill the file buffer with the first component of the file. However, because of delayed device access, an item of data is not supplied from the input device to fill the file buffer until the next reference to the file.

When writing a program for which the input will be supplied by the predeclared text file INPUT, you should be aware that delayed device access occurs. Because RESET initiates delated device access, and because EOF and EOLN cause the buffer to be filled, you should place the first prompt for input before any tests for EOF or EOLN. The information you enter in response to the prompt supplies the data that is retained by the file device until you make another reference to the input file.

Example

```
VAR
   I : INTEGER;
   .
   .
   .
BEGIN
WRITE ('Enter an integer or an empty line: ');
WHILE NOT EOLN DO
   BEGIN
   READLN (I);
   WRITELN ('The integer was: ' , I:1);
   WRITE ('Enter an integer or an empty line: ');
   END;
WRITELN ('Done');
END.
```

The first reference to the file INPUT is the EOLN test in the WHILE statement. When the test is performed, the system attempts to read a line of input from the text file. Therefore, it is very important to prompt for the integer or empty line before testing for EOLN.

Suppose you respond to the first prompt by supplying an integer as input. Access to the input device is delayed until the EOLN function makes the first reference to the file INPUT. The EOLN function causes a line of text to be read into the internal line buffer. The subsequent READLN procedure reads the input value from the line of text and assigns it to the variable I. The WRITELN procedure writes the input value to the text file OUTPUT. The final statement in the WHILE loop is the request for another input value. The loop terminates when the EOLN detects the end-of-line marker.

CHAPTER 8

USING PASCAL ON TOPS-20

This chapter describes how you use PASCAL with the TOPS-20 operating system. The steps in the program development process include:

- Creating the source program
- Compiling the program
- Loading the program
- Executing the program

This chapter describes the standard TOPS-20 file specifications and defaults, and contains instructions for creating, compiling, loading, and executing a PASCAL program.

8.1 PROGRAM DEVELOPMENT PROCESS

The TOPS-20 operating system provides a variety of methods to produce an executable program.

The first step is to create a program using an editor. This is described in Section 8.3.

The second step is to compile the program using the PASCAL command. This is described in Section 8.4.

The third step is to load the program into memory using either the LOAD command or the LINK program. The LOAD command is described in Section 8.5. For more information about LINK, refer to the LINK Reference Manual.

At this point you can use the START or the SAVE command. START runs the program that is currently loaded in memory. SAVE creates an executable image, an EXE file, and stores it in your disk area. If you use the SAVE command, you can then use the RUN command to execute the program now or at a later date.

If you do not SAVE the EXE file, you must load the file into memory before you can run it.

To save time, you can use the EXECUTE command. (See Section 8.6.) With EXECUTE, you can compile, load, and start a program all at once. EXECUTE does not create an EXE file.

8.2 FILE SPECIFICATIONS AND DEFAULTS

A file specification indicates the input file to be processed or the output file to be produced. File specifications have the following form:

```
     device:<directory>filename.filetype.gen
```

The punctuation marks (colons, angle brackets, and periods) are required syntax that separate the various components of the file specification.

device

> identifies the device or file structure on which the file is stored or is to be written.

directory

> identifies the name of the directory under which the file is catalogued, on the device specified. You can delimit the directory name with angle brackets, as shown above.

filename

> identifies the file by its name. The source file name can be up to 39 alphanumeric characters. REL file names can be up to six characters.

filetype

> describes the kind of data in the file. The source file type can be up to 39 alphanumeric characters. REL file types can be up to three characters.

gen

> specifies the generation of the TOPS-20 file desired. Generations are identified by a decimal number, which is incremented by 1 each time a new generation of a file is created. A period is used to separate file type and generation.

You need not explicitly state all elements of a file specification each time you compile, load, or execute a program. Only the file name is required, as long as you use the default file type. Table 8-1 summarizes the default values.

Table 8-1: File Specification Defaults

Optional Element	**Default Value**
device	User's current default device (DSK:)
directory	User's current default directory
file type	Depends on usage: Input to PASCAL compiler - PAS Output from PASCAL compiler - REL Input to LINK - REL Output from SAVE command - EXE Input to RUN command - EXE Compiler source listing - LST LINK map listing - MAP Input to executing program - DAT Output from executing program - DAT
gen	Input: highest existing generation Output: highest existing generation plus 1

When compiling a PASCAL program, you need specify only the file name if the file is:

- Stored on the default device
- Catalogued under the default directory name
- A file type of PAS

If more than one file meets these conditions, the compiler chooses the one with the highest generation number.

For example, assume that your default device is PS:; your default directory is <CHEN>; and you supply the following file specification to the compiler:

```
@PASCAL
PASCAL>CIRCLE
```

The compiler searches device PS: in directory <CHEN>, seeking the highest generation of CIRCLE.PAS. The compiler then generates the file CIRCLE.REL, stores it on device PS: in directory <CHEN>, and assigns it a generation number that is one higher than any other generation of CIRCLE.REL currently in PS:<CHEN>.

8.3 CREATING A PROGRAM

The first step in creating a program is to design and plan it. The TOPS-20 PASCAL Primer describes the use of PASCAL for the novice PASCAL programmer who is already familiar with higher-level programming language concepts. Many books exist that describe programming techniques, methods, and algorithms.

After planning the program, you use an editor to create a file that contains the source statements. You use a text editor to create a source file. You can use EDIT, TV, or any other text editor to create the source file.

For example, to create a PASCAL program that has the file name EXAMPL and a file type of PAS, you can issue the EDIT command as follows:

```
     @EDIT EXAMPL.PAS (RET)
     Input: EXAMPL.PAS
     00100
```

If this is a new file, an additional message is displayed indicating that a new file is being created. Because the EDIT command does not assume a file type, you must include the file type as part of the file name. The EDIT command runs the TOPS-20 default editor EDIT. The line number (00100) prompt indicates that EDIT is ready to accept input. For information on how to use EDIT, see the TOPS-20 EDIT User's Guide.

You can also use any other editor to which you have access, for example, the TV editor. To use TV, you can either type TV to TOPS-20, or you can define the logical name EDITOR: to be SYS:TV.EXE. For more information about the use of TV, refer to the TOPS-20 TV Editor Manual.

After the program is created and edited, it is ready to be compiled.

8.4 COMPILING A PROGRAM

After creating a PASCAL source program, you compile it. At compile time, you specify the source file(s) and indicate any qualifiers you wish to use.

Optionally, the compiler produces one or more object files, which are input to LINK, and one or more listing files. The listing files contain source-code listings, information about compilation errors, and optional items such as cross-reference listings.

8.4.1 The PASCAL Command

To compile a source program, specify the PASCAL command and press the RETURN key. TOPS-20 then returns the PASCAL prompt, at which point you specify the file name and any switches.

```
     @PASCAL(RET)
     PASCAL>source-filename ⟦/switch(es)⟧ (RET)
```

where:

source-filename	specifies the source file(s) containing the program or module to be compiled. If you have one program split into several source files, you can specify these source files at the same time by separating the file names with a plus sign (+). If you specify more than one source file, the files are concatenated and compiled as one program.
/switch(es)	indicates special processing to be performed by the compiler.

In many cases, the simplest form of the PASCAL command is sufficient for compilation. For other situations, however, PASCAL provides compiler commands and switches to specify special processing. PASCAL compiler commands give special instructions to the compiler. PASCAL compiler switches modify the compilation of the program.

Section 8.4.2 describes the PASCAL compiler commands, and Sections 8.4.3 and 8.4.4 describe the PASCAL compiler switches.

8.4.2 PASCAL Compiler Commands

Table 8-2 lists the commands to the compiler.

Table 8-2: PASCAL Compiler Commands

Command	**Purpose**
/EXIT	Exits from the PASCAL compiler
/HELP	Displays a help message
/RUN:	Begins execution of the specified program
/TAKE:	Takes commands from the specified command file

/EXIT

The /EXIT command exits you from the compiler and closes all files that were opened by the compiler.

/HELP

The /HELP command displays a help message.

/RUN:filespec

The /RUN: command exits you from the compiler and begins execution of the specified program. Using the /RUN: command is the same as specifying the /EXIT command to the compiler and then using the operating system command RUN to execute the specified program.

/TAKE

The /TAKE command takes commands from the specified command file. The /TAKE command recognizes the default file type CMD.

Example 1

```
@PASCAL
PASCAL>/TAKE: PLAN
PASCAL>
```

Assume that the command file PLAN.CMD contains the following:

```
PLAN.PAS /NOFLAG-NON-STANDARD /LISTING(RET)
```

The /TAKE: command causes the contents of PLAN.CMD to be executed. In this example, the source file PLAN.PAS is compiled; display of warning messages for nonstandard features is suppressed; and a listing file is generated. Make sure the command file ends with a carriage-return/line-feed. After execution of the command file, you can give another command to the compiler.

Example 2

```
PASCAL>SORTER.PAS
PASCAL>/RUN: LINK
*
```

The compiler compiles the source file SORTER.PAS, and the /RUN: command is then used to run the LINK program. The /RUN: command exits you from the compiler and causes the LINK program to begin executing. The asterisk (*) is the prompt displayed by LINK.

Example 3

```
@PASCAL
PASCAL>AVER.PAS
PASCAL>/EXIT
@
```

The EXIT command exits you from the compiler and puts you at TOPS-20 command level.

8.4.3 PASCAL Compiler Switches

Table 8-3 lists the switches you can use with the PASCAL compiler. You can specify the switches following the file name or in source code comments. This section describes the effect of each switch on a PASCAL program.

Table 8-3: PASCAL Compiler Switches

Switch	Purpose	In Source	Default
/ABORT	Causes the compiler to exist at the end of a compilation that contains errors	No	Off
/BINARY[[:filespec]]	Produces a binary object file.	No	On
/CHECK	Generates code to check for various error conditions	Yes	On
/CREF or /CROSS-REFERENCE	Produces a cross-reference listing of identifiers	Yes	Off
/DEBUG	Produces information in the object file to be used with PASDDT	No	Off
/ERROR-LIMIT:n	Stops compilation after the specified number of errors	No	30
/FLAG-NON-STANDARD	Issues warning messages for nonstandard features	Yes	On
/LISTING[[:filespec]]	Produces a source listing during compilation	Yes	Off
/MACHINE-CODE	Lists generated assembly language in source listing	Yes	Off
/NATIONAL	Turns off braces as comment characters	Yes	Off
/WARNINGS	Prints diagnostics for warning-level errors	Yes	On

/ABORT

The /ABORT switch causes the compiler to exit at the end of a compilation that contains errors. This is useful when used with the /TAKE: command. The default is /NOABORT.

/BINARY ⟦:filespec⟧

The /BINARY switch can be used when you want to specify the name of the object file. The /BINARY switch has the form:

```
     /BINARY ⟦:filespec⟧
```

If you omit the file specification, the object file defaults to the name of the last source file, the default directory, and a file type of REL. You cannot specify this switch in the source code.

You can disable this switch to suppress object code, for example, when you want to test only the source program for compilation errors.

The default is /BINARY.

/CHECK

The /CHECK switch directs the compiler to generate code to perform run-time checks. This code checks for illegal assignments to sets and subranges, out-of-range array indices and case labels, and references to NIL pointers. The system issues an error message and terminates execution if any of these conditions occur.

When this switch is disabled, the compiler does not generate code for run-time checks. The default is /CHECK.

/CROSS-REFERENCE or /CREF

The /CROSS-REFERENCE switch produces a cross-reference listing of all identifiers. The compiler generates separate cross-references for each procedure and function. To get complete cross-reference listings for a program, the switch must be in effect for all modules of the program. This switch is ignored if no listing file is being generated.

The default is /NOCROSS-REFERENCE.

You can specify this switch in the source code. Note, however, that the cross-reference listing for a portion of a procedure or function may be incomplete.

/DEBUG

The /DEBUG switch specifies that the compiler is to generate information that can be used with run-time debugging.

The default is /NODEBUG.

/ERROR-LIMIT:n

The /ERROR-LIMIT switch terminates compilation after the specified number of errors, excluding warning-level errors, have been detected. The default limit is 30 errors. If this switch is disabled, compilation continues through the entire unit. You cannot specify this switch in the source code.

The default is /ERROR-LIMIT:30.

Note that, after finding 20 errors (including warning messages) on any one source line, the compiler generates error 255, Too Many Errors On This Source Line. Compilation of the line continues, but no further error messages are printed for that line.

/FLAG-NON-STANDARD

The /FLAG-NON-STANDARD switch tells the compiler to print warning-level messages at each place where the program uses nonstandard PASCAL features.

Nonstandard PASCAL features are the extensions to the proposed ISO standard for the PASCAL language that are incorporated in PASCAL-20. Nonstandard features include VALUE declarations and the exponentiation operator. Appendix D lists all the extensions.

By default, /FLAG-NON-STANDARD is enabled.

/LISTING

The /LISTING switch produces a source listing file. It has the form:

```
     /LISTING [[:filespec]]
```

You can include a file specification for the listing file. The default file specification designates the name of the first source file, your default directory, and a file type of LST.

The compiler does not produce a listing file in interactive mode unless you specify the /LISTING switch. In batch mode, the compiler produces a listing file by default. In either case, the listing file is not automatically printed.

/MACHINE-CODE

The /MACHINE-CODE switch places in the listing file a representation of the object code generated by the compiler.

The compiler ignores this switch if the /LISTING switch is not enabled.

The default is /NOMACHINE-CODE.

/NATIONAL

The /NATIONAL switch causes the braces to have no special meaning. Therefore, if you specify the /NATIONAL switch, you cannot use braces as comment characters. Instead, you must use (* *).

The default is /NONATIONAL.

/WARNINGS

The /WARNINGS switch directs the compiler to generate diagnostic messages in response to warning-level errors.

By default, /WARNINGS is enabled. A warning diagnostic message indicates that the compiler has detected acceptable but unorthodox syntax, or has performed some corrective action. In either case, unexpected results may occur. To suppress warning diagnostic messages, disable this switch. Note that messages generated when the /STANDARD switch is enabled appear even if /WARNINGS is disabled.

8.4.4 Specifying Switches in the Source Code

You can use switches in the source code to enable and disable special processing during compilation. When specified in the source code, switches have the form:

```
     (*$switch + ,switch + ,...  ;comment *)
```

The first character after the comment delimiter must be a dollar sign ($); the dollar sign cannot be preceded by a space. Table 8-4 lists the switches you can specify in your source program. Note that you can optionally use a 1-character abbreviation for each switch. The abbreviation is simply the first character of the switch name, except for CROSS-REFERENCE, which has X for an abbreviation.

Table 8-4: Source Switches

Abbreviation	**Full**	**Command-Line Switch**
C	CHECK	CHECK
L	LIST	LIST
M	MACHINE-CODE	MACHINE-CODE
N	NATIONAL	NATIONAL
S	STANDARD	FLAG-NON-STANDARD
W	WARNINGS	WARNINGS
X	CROSS-REFERENCE	CREF

To enable a switch, specify a plus sign (+) after its name or abbreviation. To disable a switch, specify a minus sign (-) after its name or abbreviation. You can specify any number of switches. You can also include a text comment after the switches, separated from the list of switches by a semicolon.

When specified in the source code, the LIST switch cannot contain a file specification. The listing file has the default specification described above.

For example, to generate check code for only one procedure in a program, enable the CHECK switch before the procedure declaration, and disable it at the end of the procedure, as follows:

```
     (*$C+ ; enable CHECK for TEST1 only *)
     PROCEDURE TEST1;
     .
     .
     .
     END ;
     (*$C-;disable CHECK *)
```

Command line switches override source-code switches. If, for example, the source code specifies NOWARNINGS, but you type /WARN on the command line, warning messages will be generated.

NOTE

When specifying the NATIONAL switch in the source code, always use the parentheses/asterisks combination (* *) and not braces { }.

8.4.5 Specifying Output Files

The PASCAL compiler can produce object files and listing files, as well as compile the source code. You can control the production of these files with the addition of various file names and switches on the PASCAL command line.

PASCAL produces an object file automatically, taking the name from the source file and assigning it the file type REL. To change the name of the object file, specify the /BINARY switch with a file name.

To produce a listing file, you must specify the /LISTING switch on the PASCAL command line. You have the option of giving a file name with the /LISTING switch or taking the default, which is the name of the source file and the file type LST. Note, however, if you run PASCAL from a batch control file, you automatically receive a listing file. In this case, to suppress the creation of a listing file, specify the /NOLISTING switch in the batch control file.

During the early stages of program development, it is often useful to suppress the production of object files until your source program compiles without error. To suppress the production of an object file, specify the /NOBINARY switch along with the source file.

You can specify more than one source file at a time, to be concatenated and compiled. When specifying multiple source files, separate each one with a plus sign (+). Although you may specify more than one source file, you still receive one object file to load for execution. By default, the object file produced from concatenated source files has the name of the last source file on the command line. All other file specification attributes (device, directory, and so forth) assume the default attributes.

Example 1

```
@PASCAL
PASCAL>XXX+YYY+ZZZ
```

Source files XXX.PAS, YYY.PAS, and ZZZ.PAS are concatenated and compiled as one file, producing an object file named ZZZ.REL. In batch mode, this command also produces the listing file ZZZ.LST.

Example 2

```
@PASCAL
PASCAL><S.GRAVES>MNP/LISTING
```

The source file MNP.PAS in directory <S.GRAVES> is compiled, producing an object file named MNP.REL and a listing file named MNP.LST. The compiler places the object and listing files in the default directory.

8.4.6 Compiler Listing Format

When you request a listing file (by specifying the /LIST switch, PASCAL produces a compiler listing. This section explains the format of the compiler listing illustrated in Figure 8-1.

```
 ❶                                              ❷                  ❸                 ❹
AVERAGE_SCORE                ❺ 17-Aug-1983   11:23:32       PASCAL-20 1(611)      Page   1
SOURCE LISTING     CBL20:<MASELLA>AVER.PAS                                                      (1)

    LINE       ADDRESS  PROC     LEVEL
   NUMBERS   DATA INST   NO    PROC STMT  STATEMENT.     .        .        .        .        .        .        .

   100      1  000041       0      1    0   PROGRAM Average_Score (INPUT,OUTPUT);
                                                       ⓬←^456←⓭                                            ***       1 ===>     0
% PAS456  Nonstandard Pascal: "$" OR "_" in identifier in AVERAGE_SCORE⓮                                          ⓯          ⓰
   200      2  000215       0      1    0   VAR
   300      3  000215     ❾0    ❿1   ⓫0     Score, Total, Count : INTEGER;
❻ 400    ❼ 4  000220       0      1    0     AverageScore : REAL;
   500      5  000221       0      1    0
   600      6 ❽ 400000      0      1    0   BEGIN
   700      7    400012     0      1    1     Total := 0;
   800      8    400033     0      1    1     Count := 0;
   900      9    400034     0      1    1     WRITELN ('Enter your scores. When done, type CTRL/Z.');
  1000     10    400052     0      1    1     WHILE NOT EOF DO
  1100     11    400061     0      1    1       BEGIN
  1200     12    400061     0      1    2         READLN (SCORE);
  1300     13    400076     0      1    2         Total := Score + Total;
  1400     14    400101     0      1    2         Count := Count + 1;
  1500     15    400104     0      1    2       End;
  1600     16    400105     0      1    1     AverageScore := Total / Count; (*to produce real results*)
  1700     17    400113     0      1    1     WRITELN ('The average score is: ', AverageScore:4:1);
  1800     18    400142     0      1    1   END.
  1800     19    400172     0      1    0

  1 Nonstandard feature
 Last error (warning) on line      1.⓱

 Active options at end of compilation:
 NODEBUG,STANDARD,LIST,CHECK,WARNINGS,CROSS_REFERENCE,⓲
 MACHINE_CODE,OBJECT,ERROR_LIMIT =  30

Compilation time:    1.51 seconds (755 lines per minute).⓳


AVERAGE_SCORE                   17-Aug-1983   11:23:32       PASCAL-20 1(611)
GENERATED CODE     CBL20:<MASELLA>AVER.PAS                                                      (1)

   LINE    INSTRUCTION          ADDRESS              OPCODE   OPERAND(S)

 ⓴ 7     255 00 0 00 000000     400012               JFCL     00
         265 16 0 00 000000*    400013               JSP      AC16,00  PASLD%
         202 17 0 00 000002'    400014               MOVEM    AC17,02
      ㉑ 400 16 0 00 000000   ㉒ 400015            ㉓ SETZ   ㉔ AC16,
         201 05 0 00 000041'    400016               MOVEI    AC05,000041
         261 17 0 00 000005     400017               PUSH     AC17,05
         260 17 0 00 000000*    400020               PUSHJ    AC17,00  INP%IN
         105 17 0 00 777777     400021               ADJSP    AC17,777777
         200 00 0 00 000017     400022               MOVE     AC00,17
         201 05 0 00 000041'    400023               MOVEI    AC05,000041
                .
                .
                .
```

```
AVERAGE_SCORE                 17-Aug-1983   11:23:32       PASCAL-20 1(611)      Page  2
CROSS REFERENCE    CBL20:<MASELLA>AVER.PAS                                                  (1)

  AVERAGESCORE                             4      16      17
  AVERAGE_SCORE                            1
  COUNT                                    3       8      14      14      16
  INPUT [25]                               1
  OUTPUT                                   1
  SCORE                                    3      12      13
  TOTAL                                    3       7      13      13      16

 GLOBALLY DEFINED IDENTIFIERS:[26]


AVERAGE_SCORE                 17-Aug-1983   11:23:32       PASCAL-20 1(611)
GENERATED CODE     CBL20:<MASELLA>AVER.PAS                                                  (1)

   LINE   INSTRUCTION          ADDRESS              OPCODE   OPERAND(S)

          523214520302         400153               CONSTANT
          733136260716         400154               CONSTANT
          625016361736         400155               CONSTANT
          713124064746         400156               CONSTANT
          351000000000         400157 [28]          CONSTANT
     [27] 427356462744         400160               CONSTANT
          203635772744         400161               CONSTANT
          203474367744         400162               CONSTANT
          627465620256         400163               CONSTANT
          643135620310         400164               CONSTANT
          677354526100         400165               CONSTANT
          723636062500         400166               CONSTANT
          416512246136         400167               CONSTANT
          551340000000         400170               CONSTANT
          000000000000         400171               CONSTANT
                                                                               MR-S-3122-83
```

Figure 8-1: Compiler Listing Format

The compiler listing in Figure 8-1 contains the following three sections:

- Source-code listing - When you request a listing file, the source code is listed by default.
- Machine-code listing - To generate the machine-code listing, you must specify the /MACHINE-CODE switch, or the MACHINE-CODE + source switch.
- Cross-reference listing - To generate cross-references for all identifiers used in the program, you must specify the /CREF switch.

The numbers throughout this section are keyed to the numbers in Figure 8-1.

TITLE LINE - Each page of the listing contains a title line. The title line lists the module name ❶, the date and time of the compilation ❷, the PASCAL compiler name and version number ❸, the listing page number ❹, and the file specification of the source file ❺.

8.4.6.1 Source-Code Listing - The lines of the source code are printed in the source-code listing. In addition, the listing contains the following information pertaining to the source code:

- Editor line numbers ❻ - If you created or edited the source file with an editor that automatically inserts line numbers, these line numbers appear in the leftmost column of the source-code listing. Editor line numbers are irrelevant to the PASCAL compiler. Note these are not the line numbers you specify to PASDDT for debugging purposes.
- Line numbers ❼ - The compiler assigns unique line numbers to the source lines in a PASCAL module. The symbolic traceback that is printed if your program encounters an exception at run time refers to these line numbers. Note these are the line numbers you use when working with PASDDT. Refer to Chapter 9 for information concerning the debugger.
- Address ❽ - The listing includes the octal address of the instruction on that line of source code.
- Procedure number ❾ - PASCAL numbers each procedure in the listing file as it progresses down the source code, starting with 0 for the main program.
- Procedure level ❿ - Each line that contains a declaration lists the procedure level of that declaration. Procedure level 1 indicates declarations in the outermost block. The procedure-level number increases by one for each nesting level of functions or procedures.

- Statement level ⓫ - The listing specifies a statement level for each line of source code after the first BEGIN delimiter. The statement level starts at 0 and increases by 1 for each nesting level of PASCAL structured statements. Specifically, the CASE statement, the REPEAT statement, and the BEGIN/END block increase the statement level. The statement level of a comment is the same as that of the statement that follows it.

ERRORS and WARNINGS -- The source-code listing includes information on any errors or warnings detected by the compiler. The actual message is printed beneath the error in the source code. In addition, the following items appear in the listing:

- A circumflex (^) that points to the character position in the line where the error was detected ⓬ .

- A numeric code, following the circumflex, that specifies the particular error ⓭ . On the following lines of the source listing, the compiler prints the text that corresponds to each numeric code ⓮ .

- The line number where the error was detected ⓯ and the line number of the previous line containing an error ⓰ . You can use these error line numbers to trace the error diagnostics back through the source listing.

SUMMARY -- At the end of the source listing, the compiler tells you how many errors or warnings were generated (if any), with the source line number where the last one occurred ⓱ . The compiler prints the status of all the compilation options ⓲ and how much time was required for the compilation ⓳ .

8.4.6.2 Machine-Code Listing

8.4.6.2 Machine-Code Listing - The machine-code listing (if requested with the /MACHINE-CODE switch) follows the source-code listing. The machine-code listing contains:

- Source line number ⓴ - A source line number marks the first object instruction that the compiler generated for the first PASCAL statement on that source line.

- Octal representation of instruction ㉑ - This is the octal representation of the object instruction.

- Address ㉒ - This is the relocatable address of the instruction.

- Opcode ㉓ - This is the mnemonic operation code for the instruction.

- Operand ㉔ - This contains the mnemonic accumulator field and address field for the operation.

For more information on machine-code instructions, refer to the DECsystem-10/DECSYSTEM-20 Processor Reference Manual.

8.4.6.3 Cross-Reference Listing - The cross-reference listing (if requested with the /CREF switch) appears after the machine-code listing. It contains two sections:

- User-specified identifiers ㉕ - This section lists all the identifiers you declared.
- Globally-defined identifiers ㉖ - This section lists the PASCAL predefined identifiers that the program uses.

Each line of the cross-reference listing contains an identifier and a list of the source lines where the identifier is used ㉗ . The first line number indicates where the identifier is declared. Predefined identifiers are listed as if they were declared on line 0 ㉘ . The cross-reference listing does not specify pointer type identifiers that are used before they are declared.

8.5 LOADING A PROGRAM

After compiling your PASCAL program, you load the object module(s) with the LINK program to produce an executable image file. Loading resolves all references in the object code and establishes absolute addresses for symbolic locations. This section describes the use of the LOAD command.

8.5.1 The LOAD Command

To load an object module, specify the LOAD command in the following general form:

```
     LOAD filename  ⟦/switch(es)...⟧
```

where:

filename	specifies the input object file to be loaded. The file must be a REL file, created with the PASCAL compiler (or some other translator).
/switch(es)	specify input file options.

The file specification must be typed on the same line as the LOAD command. If the file specification does not fit on one line, you can continue typing without pressing the RETURN key.

The LOAD command runs the LINK program, which reads the REL file specified on the command line and loads it into memory. When LINK exits, it leaves your program in memory, ready to be SAVEd as an EXE file or to be STARTed.

The LOAD command can do a COMPILE command before running LINK. If you specify a file name to the LOAD command for which there is a source file but no REL file, the LOAD command automatically performs a COMPILE command first.

See the <u>TOPS-20 Commands Reference Manual</u> for more information on the LOAD command.

8.6 EXECUTING A PROGRAM

After you have compiled and linked your program, you can execute the object file with the START command; or you can save the file with the SAVE command, and then execute it with the RUN command.

You can save time by using the EXECUTE command. The EXECUTE command acts by performing a LOAD command (which may start with a COMPILE command) followed immediately by a START command. So, instead of performing several separate steps of running the compiler, compiling your program, loading your program, and starting your program, you can compile, load, and start your program with a single command:

```
     @EXECUTE filename RET
```

Using the EXECUTE command does not save the executable image of the object file. The SAVE command stores a copy of the executable image in an executable file. The default file type that is created is EXE. After an executable image is saved, you can execute it with the RUN command. The SAVE and RUN commands have the form:

```
     @SAVE filename RET
     @RUN filename RET
```

You must specify the file name; default values are applied if you omit optional elements of the file specification. The default file type is EXE.

8.7 EXAMPLES

Example 1

```
     @PASCAL
     PASCAL>SORTER /NOFLAG/LISTING
     PASCAL>/EXIT
     @EXE SORTER
```

This example uses the PASCAL command to compile the source file SORTER. The /NOFLAG switch prevents display of messages about nonstandard PASCAL statements in the program; the /LISTING switch generates a .LST file of the compilation in your disk area.

Example 2

```
     @PASCAL
     PASCAL>MERGER/DEBUG
     PASCAL>/EXIT
     @LOAD MERGER/DEBUG
     @DEBUG MERGER
```

The PASCAL command is used to compile the program with the debugger, PASDDT. The DEBUG command is used to load the compiled program and the PASCAL debugger automatically. The DEBUG command executes the program with PASDDT.

Example 3

```
     @LOAD PLAN /COMPILE /CREF
     @PRINT PLAN.LST
     @START
```

The LOAD command first compiles the program because of the /COMPILE switch, then, if compilation is successful, loads the program. The /CREF switch generates a listing file with cross-reference information. The PRINT command prints the listing. The program is then executed with the START command.

CHAPTER 9

PASDDT: THE PASCAL-20 DEBUGGER

The PASCAL-20 debugger, PASDDT, provides the means to monitor and modify the execution of a PASCAL program. PASDDT provides symbolic debugging capabilities that allow you to read and modify the values associated with variables by referring to the PASCAL identifiers within your program.

9.1 RUNNING PASDDT

To use the debugger, you must compile and load PASDDT with the program. First, compile the source program and include the /DEBUG switch in the command string:

```
     @PASCAL
     PASCAL>filename/DEBUG
     PASCAL>/EXIT
     @
```

Then, load the program along with PASDDT by specifying the DEBUG command:

```
     @DEBUG filename
```

The DEBUG command loads and starts the program currently in memory, with the debugger, PASDDT.

If you want more control over where the debugger is placed, or you have the need for more options, you can use the TOPS-20 LINK program directly. Refer to the LINK Reference Manual for TOPS-20 information.

When running PASDDT, the source file with the extension PAS should be located in the same directory as the EXE version of the file. This should also be the directory to which you are connected when running PASDDT. If you have specified a file with the %INCLUDE directive, PASDDT looks for a file with the same name.

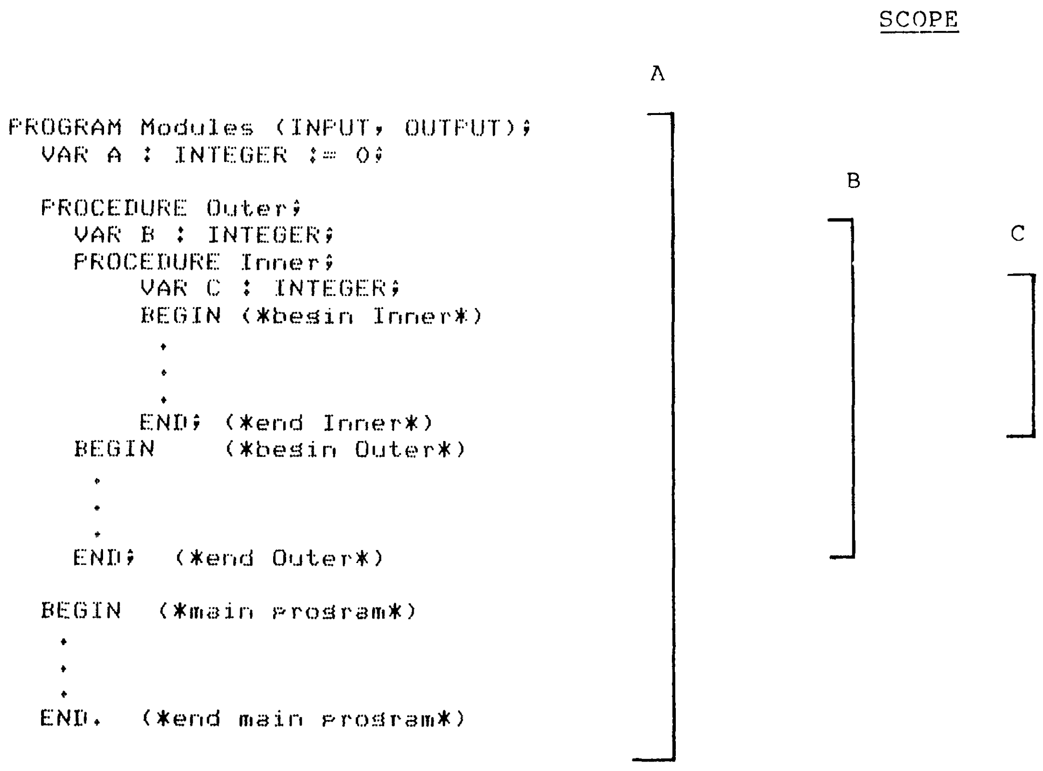
9.2 USING SYMBOLIC VALUES

Symbolic values are the identifiers defined within the source program. PASDDT allows access to identifiers available only in the current scope, and performs recognition on these identifiers.

You can specify a location in the source code by using the line number shown in the listing file created when the program is compiled. See Section 8.4.7.

9.3 SCOPE

Scope is the range within the program in which a specific definition of an identifier exists. The scope corresponds to the part of the program in which the identifier can be used. Figure 9-1 shows the scope of three variables in program Modules. The scope of the identifier A (a global variable) is the entire program. The scope of variable B, declared in procedure Outer, is the entire procedure, including procedure Inner. The scope of variable C is limited to procedure Inner.



```
PROGRAM Modules (INPUT, OUTPUT);
  VAR A : INTEGER := 0;

  PROCEDURE Outer;
    VAR B : INTEGER;
    PROCEDURE Inner;
        VAR C : INTEGER;
        BEGIN (*begin Inner*)
          .
          .
          .
        END; (*end Inner*)
    BEGIN     (*begin Outer*)
     .
     .
     .
    END;   (*end Outer*)

  BEGIN   (*main program*)
   .
   .
   .
  END.  (*end main program*)
```

Figure 9-1: Scope

PASDDT uses the concept of dynamic scope when accessing identifiers and their values. Dynamic scope is the scope in relation to the use of the program; the dynamic scope refers to the level at which a particular identifier is being used. For example, a recursive procedure (a procedure that calls itself) has multiple scopes for the same identifier.

PASDDT represents the scope of an identifier with a positive integer. Global data has a scope of 1. Each nested level from that has a scope of one greater than that of the level from which it is called. The following example is a recursive function that calculates factorials.

```
PROGRAM Calculate (INPUT,OUTPUT);

     VAR Answer, Num : INTEGER;

     FUNCTION Factorial (Number : INTEGER) : INTEGER;
       BEGIN
          WHILE Number > 0 DO
             Factorial := Number * Factorial (Number - 1);
       END;

     BEGIN
        Answer := Factorial(Num);
        .
        .
        .
     END.
```

When Factorial is called from the main program, the value of Num is passed to Number. Until Number is equal to 0, the function Factorial continues to call itself; the value of Number is decremented by 1 each time the function calls itself. Assume that the value 3 is passed to Number. Number then has the value of 3, then of 2, then of 1, then of 0. Each call to itself represents another level of dynamic scope. Number has a scope of 1 the first time it is called; a scope of 2 the second time it is called, and so on.

9.4 PASDDT COMMANDS

The following sections describe each of the PASDDT commands. The default radix for all purposes is decimal.

PASDDT uses the facility of recognition and guidewords. Recognition permits you to type enough of the word or identifier to be unique. For example, you can type CL to specify the CLEAR command. You can also use recognition with identifiers you have used in the program. For example, if you have defined the identifiers NewList and NewTABLE, you could use NewL and NewT, respectively, to specify these identifiers.

PASDDT also provides guidewords when you press the ESC key. The guide words indicate what you should enter next. In the following sections, guidewords are displayed in parentheses. You do not need to type the guidewords or the surrounding parentheses in the syntax. In addition, PASDDT displays the options that you can enter when you type a question mark (?). For more information about recognition and guidewords, refer to the TOPS-20 User's Guide.

9.4.1 ASSIGN

ASSIGN assigns a value or virtual address to the specified identifier. The format is:

$$\text{ASSIGN (VARIABLE or ADDRESS)} \begin{Bmatrix}\text{user identifier}\\ \text{octal address}\end{Bmatrix} \quad (:=) \quad \begin{Bmatrix}\text{constant}\\ \text{octal value}\end{Bmatrix}$$

where:

user identifier	is a variable name in the active scope.
octal address	is a virtual address in the user program.

constant	is a value with a simple data type of INTEGER, REAL, CHAR, BOOLEAN, DOUBLE, or user-defined enumerated type.
octal value	is the value to be placed in the octal address.

Example

```
PASDDT>ASSIGN New_Int (:=) 20
```

This example assigns the value of 20 to user identifier New_Int.

9.4.2 BREAK

BREAK sets a new breakpoint at a specified location or resets an existing breakpoint. During a debugging session, when the program reaches the location specified, it stops execution, thus allowing you to perform debugging operations. You can set a maximum of 20 breakpoints.

To set a new breakpoint, use the following format of the BREAK command:

```
BREAK (AT) line number [[(NAME) break identifier]]
```

where:

line number	is a line number corresponding to a line in the source code. These line numbers are found in the listing file generated when you compile the program.
break identifier	is the name you associate with the breakpoint. Each breakpoint identifier can contain any number of characters, but the first nine characters must be unique. The break identifier can contain all characters except an underscore (_) or dollar sign ($).

Note that you cannot set a breakpoint in the declaration section of a program. If you do, PASDDT sends you an error message and does not set the breakpoint.

To change the status of a breakpoint from CLEAR (Section 9.4.3), use the following format of the BREAK command:

```
BREAK (AT) {line number     }
           {break identifier}
```

where:

line number	is the line number of the breakpoint that you want to reactivate.
break identifier	is the name of the breakpoint you want to reactivate.

Note that you can find out the status of breakpoints by displaying them with the DISPLAY command. See Section 9.4.4 for information on the DISPLAY command.

The following rules apply to the use of BREAK:

If you specify a break identifier, but not a line number, the following happens:

- If there is already a breakpoint with this identifier, this breakpoint is used.
- If there is no breakpoint with this identifier, an error occurs.

If you specify a line number, but not a break identifier, the following happens:

- If a breakpoint is assigned for this location, that breakpoint is used.
- If a breakpoint has not been assigned, a new breakpoint is established with the identifier NO_NAME.

If the command specifies both a line number and a break identifier, the following happens:

- If a breakpoint has already been assigned for this location with this identifier, the previously defined breakpoint is used.
- If a breakpoint has already been assigned for this location with a different identifier, an error occurs.
- If a breakpoint has already been assigned at another location with this identifier, and no breakpoint has been set at this location, PASDDT prints a warning message. PASDDT then asks if you want to override the address associated with that breakpoint. If so, the previous breakpoint is discarded, and a new breakpoint is declared with the given identifier and location.

Note that you must use the PROCEED command after each break for the program to continue. PROCEED is described in Section 9.4.7.

Note the following example:

```
PASDDT>break 10 first
PASDDT>display break
BREAK POINT(S):

  Name          = FIRST
  Break Address  =  000000400044
  Line Number    =          10
  Status            =  BREAK
PASDDT>proceed
>>BREAK:FIRST
PC AT  VIRTUAL ADDRESS      00000040044
PASCAL LINE NUMBER     =           10
LINE 9:    Count := 0;
LINE 10:     WRITELN ('Enter your scores. When done, type CTRL/Z.');
LINE 11:     WHILE NOT EOF DO
PASDDT>
```

As shown in the previous example, PASDDT automatically displays three lines of source code at a breakpoint. See the SET command, Section 9.4.9, for details of controlling the display of source code when a breakpoint is reached.

9.4.3 CLEAR

CLEAR turns off breakpoints and tracepoints set with the BREAK and TRACE commands. The format is:

```
CLEAR (AT) {line number     } [[(NAME) break identifier]]
           {break identifier}
```

where:

line number	is the line number in the source code where the breakpoint or tracepoint is set.
break identifier	is the name of the breakpoint or tracepoint associated with the line number.

You can specify either both the line number and associated break identifier, just the line number, or just the break identifier.

The CLEAR command does not delete the breakpoint or tracepoint; the CLEAR command merely turns it off. Also, although you have cleared a breakpoint or tracepoint, it is still counted in the total 20 allowed. To delete a breakpoint or tracepoint, use the REMOVE command, Section 9.4.8. If you want to check the status of breakpoints and tracepoints, use the DISPLAY command, Section 9.4.4.

Note the following example:

```
PASDDT>break 11 one
PASDDT>break 12 two
PASDDT>display break
BREAK POINT(S):

  Name          = ONE
  Break Address  =  000000400062
  Line Number    =          11
  Status              =  BREAK

  Name          = TWO
  Break Address  =  000000400071
  Line Number    =          12
  Status              =  BREAK
PASDDT>clear 11
PASDDT>display break
BREAK POINT(S):

  Name          = ONE
  Break Address     =  000000400062
  Line Number       =          11
  Status                =    OFF

  Name          = TWO
  Break Address     =  000000400071
  Line Number       =          12
  Status                =  BREAK
```

```
PASDDT>clear two
PASDDT>display break
BREAK POINT(S):

   Name         = ONE
   Break Address  =  000000400062
   Line Number    =          11
   Status              =     OFF

   Name         = TWO
   Break Address  =  000000400071
   Line Number    =          12
   Status              =     OFF
PASDDT>break 12
PASDDT>display break
BREAK POINT(S):

   Name         = ONE
   Break Address  =  000000400062
   Line Number    =          11
   Status              =     OFF

   Name         = TWO
   Break Address  =  000000400071
   Line Number    =          12
   Status              =   BREAK
PASDDT>ex
```

9.4.4 DISPLAY

The DISPLAY command has three functions:

1. To show the status of breakpoints and tracepoints

2. To show the calling sequence of procedures and functions

3. To show a line (or range of lines) of the source code

To DISPLAY information about breakpoints and tracepoints, use the following format:

DISPLAY B ⟦REAK-IDENTIFIER⟧   ⟦break identifier⟧

where:

B ⟦REAK-IDENTIFIER⟧	indicates that you want to display breakpoints and tracepoints.
break identifier	is the name of a particular breakpoint or tracepoint. If you do not specify a break identifier, PASDDT displays information about all of them.

The information you receive includes the name of the breakpoint, the line number of the breakpoint, and the status of the breakpoint (ON or OFF).

Note the following example:

```
PASDDT>display break
BREAK POINT(S):

  Name          = FIRST
  Break Address  =  000000000375
  Line Number     =           19
  Status             =  BREAK

  Name          = SECOND
  Break Address  =  000000000452
  Line Number     =           23
  Status             =  TRACE

  Name          = THIRD
  Break Address  =  000000000617
  Line Number     =           37
PASDDT>display break second
BREAK POINT(S):

  Name          = SECOND
  Break Address  =  000000000452
  Line Number     =           23
  Status             =  TRACE
```

To DISPLAY the calling sequence of procedures and functions, use the following format:

```
DISPLAY (OPTION) I [[NVOCATION-STACK]] {integer} [[{/STATIC }]]
                                       {   *   }   {/DYNAMIC}
```

where:

I [[NVOCATION-STACK]]	indicates that you want to display the calling sequence of your routines.
integer	is a decimal number representing how many routine calls to display.
*	indicates that you want all routine calls displayed back to the main program.
/STATIC	causes the static level of the routines to be displayed as they were defined in the user program.
/DYNAMIC	causes every invocation of a routine to be displayed. This is the default.

The information you receive includes the calling sequence, the static level of the procedure or function, and the address at the time of the display.

Note the following example:

```
PASDDT>display invocation-stack */static

PROCEDURE NAME                          STATIC LEVEL   ADDRESS
GROCERY_BILL                                 1          000140
PASDDT>display invoc */dynamic

PROCEDURE NAME                          STATIC LEVEL   ADDRESS
GROCERY_BILL                                 1          000140
PASDDT>
```

To DISPLAY lines of source code in the current scope, use the following format:

DISPLAY (OPTION) S ⟦OURCE-LINE⟧ line number ⟦/RANGE:integer⟧

where:

S ⟦OURCE-LINE⟧	indicates that you want to display a line of the source code.
line number	is an integer representing a line in the source code. You will find these numbers in the listing file.
/RANGE:	is an optional value indicating that you want to display more than one line of source code.
integer	is a decimal number indicating how many lines of source code to display.

DISPLAY prints the source code of the line number you specify and any other lines included in the range.

Note the following example:

```
PASDDT>display source 5/range:10
LINE 5:      Yes_No = (Yes , No);  (* Defines data type Yes_No
LINE 6:                               with values Yes and No *)
LINE 7:
LINE 8: VAR
LINE 9:      Item_Price , Total,
LINE 10:      Coupon_Amount : Real;  (*Declares three real variables*)
LINE 11:      Ans : Yes_No;   (* Declares a variable, Ans, of type
LINE 12:                          Yes_No *)
LINE 13:      Subtotal , Coupons: REAL := 0.0;
LINE 14:
PASDDT>
```

9.4.5 EXIT

EXIT halts execution of the program. The format is:

EXIT

Note the following example:

```
     PASDDT>EXIT

     CPU time:  00.16 Elapsed time:  03.37
     @
```

This exits you from PASDDT and puts you at TOPS-20 command level. The CPU time and elapsed time are also displayed.

9.4.6 HELP

HELP displays a help message. The format is:

    HELP (with PASDDT) [[command name]]

Note the following example:

```
     PASDDT>help

[HELP]               HELP [command name or return]

                     This command allows the user to get help with  PASDDT.
                     Type  'HELP'  followed  by  the  command you want help
                     with.  Type a question mark ('?') to see the  commands
                     that are available.

PASDDT>ex
```

This displays help messages about all of the PASDDT commands.

9.4.7 PROCEED

PROCEED initiates or continues execution of the program until a breakpoint is reached or the user program terminates. The format is:

    PROCEED (with USER Program)

Example 1

```
     @PASCAL
     PASCAL>TEST.PAS/DEBUG
     PASCAL>/EXIT
     @DEBUG TEST
     PASDDT>PROCEED
```

This example shows source program TEST.PAS being compiled, loaded, and started. PROCEED initiates the execution of the program.

Example 2

```
     @PASCAL
     @PASCAL>EXAMPL /DEBUG
     PASCAL>/EXIT
     @LOAD EXAMPL.REL,SYS:PASDDT.REL
     @SAVE EXAMPL
     @RUN EXAMPL
     PASDDT>PROCEED
```

This example compiles and loads the program EXAMPL. Loading PASDDT.REL with the program EXAMPL causes a copy of PASDDT to become a part of the executable file EXAMPL.EXE. PROCEED initiates the execution of the program.

Example 3

```
     PASDDT>BREAK 35 Break1

     PASDDT>PROCEED
```

This example causes the program to resume execution after stopping at a breakpoint.

9.4.8 REMOVE

REMOVE deletes breakpoints and tracepoints completely. The format is:

```
     REMOVE (AT) {line number     }   [[(NAME) break identifier]]
                 {break identifier}
```

where:

line number	is the line number in the source code where you set the breakpoint or tracepoint.
break identifier	is the name of the breakpoint or tracepoint.

You can specify either a line number followed by the break identifier associated with that line number, just the line number, or just the break identifier.

Note the following example:

```
PASDDT>break 10
PASDDT>display break
BREAK POINT(S):

  Name          = <NONAME>
  Break Address  =  000000400044
  Line Number    =          10
  Status              = BREAK
PASDDT>remove 10
PASDDT>display break
NO BREAKPOINTS SET
PASDDT>ex
```

9.4.9 SET

The SET command has four functions:

1. Turning on automatic displaying of source code when a breakpoint is executed. This is the default.

2. Telling PASDDT how many lines to display when a breakpoint is executed. The default is three.

3. Telling PASDDT which separately compiled module to use in terms of setting breakpoints and gaining information about the user program. When PASDDT starts up, the default is the main program.

4. Setting the level of information displayed by PASDDT. The default is VERBOSE.

To SET the automatic displaying of source code on or off, use the following format:

```
SET (OPTION) { A [[UTO-DISPLAY]]     }
             { NO-A [[UTO-DISPLAY]]  }
```

where:

AUTO-DISPLAY	turns on the automatic displaying of source code at a breakpoint. This is the default.
NO-AUTO-DISPLAY	turns off the automatic displaying of source code at a breakpoint.

When AUTO-DISPLAY is set, PASDDT displays the line of source code at the breakpoint. If your terminal is slow or you are using a dial-up line, you may want to turn off displaying the source code with NO-AUTO-DISPLAY.

Note the following example:

```
PASDDT>break 10
PASDDT>set no-auto
PASDDT>proceed
>>BREAK:
PC AT  VIRTUAL ADDRESS      000000000416
PASCAL LINE NUMBER     =            10
PASDDT>break 12
PASDDT>set auto
PASDDT>proc
Enter your scores. When done, type CTRL/Z.
23
>>BREAK:
PC AT  VIRTUAL ADDRESS      000000000443
PASCAL LINE NUMBER     =            12
LINE 11:    WHILE NOT EOF DO
LINE 12:        BEGIN
LINE 13:             READLN (Score);
```

To SET the number of lines to automatically display, use the following format of the SET command:

```
SET (OPTION) W [[INDOW]]  n
```

where:

WINDOW	indicates that you want to set the number of lines of source code to display at a breakpoint.
n	is a decimal integer specifying how many lines of source code to display. The default is three.

The window is centered around the line on which the breakpoint is set.

Note the following example:

```
PASDDT>set window 10
PASDDT>proc
45
>>BREAK:
PC AT  VIRTUAL ADDRESS       000000000443
PASCAL LINE NUMBER       =           12
LINE 7: BEGIN
LINE 8:     Total := 0;
LINE 9:     Count := 0;
LINE 10:     WRITELN ('Enter your scores. When done, type CTRL/Z.');
LINE 11:     WHILE NOT EOF DO
LINE 12:        BEGIN
LINE 13:            READLN (Score);
LINE 14:                    Total := Score + Total;
LINE 15:            Count := Count + 1;
LINE 16:        End;
```

To SET the name of the module where PASDDT can find information relevent for debugging purposes, use the following format of the SET command:

SET (OPTION) M ⟦ODULE⟧ module name

where:

MODULE	indicates that you want to specify the name of a separately compiled module.
module name	is the identifier after the word PROGRAM or MODULE in the source file. At start-up, the default is the main program.

Note the following example:

```
PASDDT>set module ? MODULE NAME AVERAGE_SCORE
PASDDT>set module averase_score
PASDDT>
```

To SET the level of information displayed by PASDDT, use the following format of the SET command:

SET (OPTION) V ⟦ERBOSITY⟧ (OF TYPEOUT) $\left\{\begin{matrix} \text{V}\left[\!\left[\text{ERBOSE}\right]\!\right] \\ \text{B}\left[\!\left[\text{RIEF}\right]\!\right] \end{matrix}\right\}$

where:

VERBOSITY	indicates that you want to change the level of information displayed.
VERBOSE	displays the most information. This is the default.
BRIEF	displays a minimal amount of information.

Note the following example:

```
PASDDT>set verbosITY (OF TYPEOUT) brief
PASDDT>proce
56
>>BREAK:
PASCAL LINE NUMBER     =          12
LINE 7: BEGIN
LINE 8:    Total := 0;
LINE 9:    Count := 0;
LINE 10:    WRITELN ('Enter your scores. When done, type CTRL/Z.');
LINE 11:    WHILE NOT EOF DO
LINE 12:       BEGIN
LINE 13:           READLN (Score);
LINE 14:                   Total := Score + Total;
LINE 15:           Count := Count + 1;
LINE 16:       End;
PASDDT>set window 5
PASDDT>proc
78
>>BREAK:
PASCAL LINE NUMBER     =          12
LINE 10:    WRITELN ('Enter your scores. When done, type CTRL/Z.');
LINE 11:    WHILE NOT EOF DO
LINE 12:       BEGIN
LINE 13:           READLN (Score);
LINE 14:                   Total := Score + Total;
PASDDT>exit
```

9.4.10 SHOW

The SHOW command prints the current value of an identifier. The format of the SHOW command is:

SHOW (VARIABLE OR ADDRESS) $\left\{\begin{array}{l}\text{user identifier}\\ \text{octal address}\end{array}\right\}$

where:

user identifier	is the name of a variable in the current scope.
octal address	is a virtual address in the user program. (This is not the address shown in the listing file.)

Note that, when you set a breakpoint, PASDDT automatically displays three lines of source code, unless you specified SET NO-AUTO-DISPLAY. You can then see any identifiers that are declared for the current scope of the user program with the SHOW command. You can also type a question mark (?) to the SHOW command to see what identifiers are available.

Note the following example:

```
PASDDT>break 10
PASDDT>proceed
>>BREAK:
PC AT  VIRTUAL ADDRESS       000000400044
PASCAL LINE NUMBER     =           10
LINE 9:     Count := 0;
LINE 10:     WRITELN ('Enter your scores. When done, type CTRL/Z.');
LINE 11:     WHILE NOT EOF DO
PASDDT>show ? VIRTUAL ADDRESS
  or PASCAL VARIABLE one of the following:
 AVERAGESCORE               AVERAGE_SCORE    COUNT       INPUT
 OUTPUT                      SCORE     TOTAL
PASDDT>show score
VALUE                        =            0
TYPE      =    INTEGER
ADDRESS   =   037060
PASDDT>show count
VALUE                        =            0
TYPE      =    INTEGER
ADDRESS   =   037062
PASDDT>show input
File is OPEN for reading
File is a TEXT file
File is TTY:
File is the standard INPUT file

File name = TTY:
PASDDT>ex
```

9.4.11 TRACE

TRACE sets a new tracepoint at a specified location or resets an existing tracepoint. When the program reaches the location specified, PASDDT prints a message indicating that a trace was placed on this line. Unlike the BREAK command, TRACE does not halt program execution or print source code. You can set a total of 20 tracepoints and breakpoints. Clearing them does not delete them from the count.

To set a new tracepoint, use the following format of the TRACE command:

```
    TRACE (AT) line number  [[(NAME) trace identifier]]
```

where:

line number	is the line number associated with the line of source code that you want to trace. This line number can be found in the listing file.
trace identifier	is the name you want to associate with the tracepoint. Each identifier can contain any number of characters, but the first nine characters must be unique. The identifier can contain all characters except underscore (_) and dollar sign ($).

To reset a tracepoint that you have turned off with CLEAR (Section 9.4.3), use the following format of the TRACE command:

```
TRACE (AT) {line number     }
           {break identifier}
```

where:

line number	is the line number of an existing tracepoint that you want to reset.
break identifier	is the name of an existing tracepoint that you want to reset.

The following rules apply to the use of TRACE:

If you specify a trace identifier, but not a line number, the following happens:

- If there is already a tracepoint with this identifier, this tracepoint is used.
- If there is no tracepoint with this identifier, an error occurs.

If you specify a line number, but not a trace identifier, the following happens:

- If a tracepoint is assigned for this location, that tracepoint is used.
- If a tracepoint has not been assigned, a new tracepoint is established with the identifier NO_NAME. You can have more than one tracepoint with the name NO_NAME.

If you specify both a line number and a trace identifier, the following happens:

- If a tracepoint has already been assigned for this location with this identifier, the previously defined tracepoint is used.
- If a tracepoint has already been assigned for this location with a different identifier, an error occurs.
- If a tracepoint has already been assigned at another location with this identifier, and no tracepoint has been set at this location, PASDDT prints a warning message. PASDDT then asks if you want to override the address associated with the previous tracepoint. If so, the previous tracepoint is discarded, and a new tracepoint is declared with the given identifier and location.

Note the following example:

```
PASDDT>trace 10 one
PASDDT>proceed
>>TRACE: ONE
PC AT  VIRTUAL ADDRESS      000000400044
PASCAL LINE NUMBER      =           10
Enter your scores. When done, type CTRL/Z.
75
44
89
^ZThe average score is: 69.3
```

APPENDIX A

PASCAL MESSAGES

Table A-1: Run-time Errors

? PRT001	Value not within subrange of variable in assignment.
? PRT002	Case selector out of range.
? PRT003	Array index out of bounds.
? PRT004	Conformant array index out of bounds.
? PRT005	Size of conformant arrays incompatible.
? PRT006	NIL pointer value at runtime.
? PRT007	Attempt to divide by 0.
? PRT008	Mod with 0 or negative value.
? PRT009	Set value out of range in assignment.
? PRT010	Set element out of range in assignment.
? PRT011	Stack expansion failed - No more memory available for stack space.
? PRT012	Memory expansion failed - No more memory available.
? PRT013	DISPOSE called with NIL pointer.
? PRT014	Page creation failed, error code = number.
? PRT015	Page destroy failed, error code = number.
? PRT016	Insufficient initial memory - cannot start program.
? PRT017	Fatal error - illegal instruction detected at user PC number.
? PRT018	Fatal error - illegal memory reference detected at user PC number.
? PRT019	Fatal error - illegal memory read detected at user PC number.
? PRT020	Fatal error - illegal memory write detected at user PC number.
? PRT021	Fatal error - non-existent page detected at user PC number.

Note that errors PRT017 through PRT021 are probably caused by improper use of an array or pointer variable. If you receive one of these errors, try compiling the program with the /CHECK switch.

Table A-2: I/O Errors

? PIO001	User buffer overflow. File in error: filespec
? PIO002	Line limit exceeded. File in error: filespec
? PIO003	File not open for reading. File in error: filespec
? PIO004	File not open for writing. File in error: filespec
? PIO005	Field width <= zero. File in error: filespec
? PIO006	String write error. File in error: filespec
? PIO007	Integer write error. File in error: filespec
? PIO008	Field width < zero. File in error: filespec
? PIO009	Attempt to read past EOF. File in error: filespec
? PIO010	Integer read error. File in error: filespec
? PIO011	String read error. File in error: filespec
? PIO012	Illegal character in number I/O. File in error: filespec
? PIO013	Attempt to RESET(output). File in error: filespec
? PIO014	Attempt to RESET(input). File in error: filespec
? PIO015	Integer overflow. File in error: filespec
? PIO016	Attempt to reset/rewrite uninitialized file. File in error: filespec
? PIO017	Error in opening binary file. File in error: filespec
? PIO018	Error in writing to object file. File in error: filespec
? PIO019	Error in closing binary file. File in error: filespec
? PIO020	Delete file error. File in error: filespec
? PIO021	Include/Exclude file error. File in error: filespec
? PIO022	Attempt to use FIND on text file. File in error: filespec
? PIO023	Attempt to FIND a negative record. File in error: filespec
? PIO024	Field width too small for number output. File in error: filespec

Table A-2: I/O Errors (Cont.)

? PIO025	Scalar out of range. File in error: filespec
? PIO026	Attempt to open/close INPUT file. File in error: filespec
? PIO027	Attempt to close OUTPUT file. File in error: filespec
? PIO028	Attempt to write to READONLY file. File in error: filespec
? PIO029	Attempt to open already opened file. File in error: filespec
? PIO030	Attempt to REWRITE(output). File in error: filespec
? PIO031	Attempt to REWRITE(input). File in error: filespec
? PIO032	Exponent too large. File in error: filespec
? PIO033	Exponent too small. File in error: filespec
? PIO034	Reset error -- File not found. File in error: filespec
? PIO035	Scalar is all blanks. File in error: filespec
? PIO036	Attempt to read past end of line. File in error: filespec
? PIO037	Attempt to readln/writeln to a binary file. File in error: filespec
? PIO040	PAGE called with file opened NOCARRIAGE. File in error: filespec
? PIO042	EOLN called when EOF true. File in error: filespec
? PIO043	Testing EOF on unopened file. File in error: filespec
? PIO044	Too many open files.
? PIO045	Attempt to RESET new internal file. File in error: filespec

Compile-Time Errors

? PAS001 Error in simple type

The declaration for a base type of a set or the index type of an array contains a syntax error.

? PAS002 Identifier expected

The statement syntax requires an identifier, but none can be found.

? PAS003 "PROGRAM" or "MODULE" expected

The statement syntax requires the reserved word PROGRAM or MODULE.

? PAS004 ")" expected

The statement syntax requires the right-parenthesis character.

? PAS005 ":" expected

The statement syntax requires a colon character.

? PAS006 Illegal symbol

The statement contains an illegal symbol, such as a misspelled reserved word or illegal character.

? PAS007 Error in parameter list

The parameter list contains a syntax error, such as a missing comma, colon, or semicolon character.

? PAS008 "OF" expected

The statement syntax requires the reserved word OF.

? PAS009 "(" expected

The statement syntax requires the left-parenthesis character.

? PAS010 Error in type

The statement syntax requires a data type, but no type identifier is present.

? PAS011 "[" expected

The statement syntax requires the left square bracket character.

? PAS012 "]" expected

The statement syntax requires the right square bracket character.

? PAS013 "END" expected

The compiler cannot find the delimiter END, which marks the end of a compound statement, subprogram, or program.

? PAS014 ";" expected

The statement syntax requires the semicolon character.

? PAS015 Integer expected

The statement syntax requires an integer.

? PAS016 "=" expected

The statement syntax requires the equal sign to separate a constant identifier from a constant value or to separate a type identifier from a type definition.

? PAS017 "BEGIN" expected

The compiler cannot find the delimiter BEGIN, which marks the beginning of an executable section.

? PAS018 ".." expected

The compiler cannot find the .. symbol, which is required between the endpoints of a subrange.

? PAS019 Error in field list

The field list in a record declaration contains a syntax error.

? PAS020 "," expected

The statement syntax requires a comma.

? PAS021 Empty parameter (successive ",") not allowed

The parameter list attempts to specify a null or missing parameter, or contains an extra comma.

? PAS022 Illegal (nonprintable) ASCII character

The program contains an illegal character that is not a printable ASCII character.

? PAS023 "," or ")" expected

The statement syntax requires either a comma or a right-parenthesis character.

? PAS024 "''" expected

The statement syntax requires two quotation marks.

? PAS050 Error in constant

A constant contains an illegal character or is improperly formed.

? PAS051 ":=" expected

The statement syntax requires the assignment operator.

? PAS052 "THEN" expected

The compiler cannot find the reserved word THEN to complete the IF-THEN statement.

? PAS053 "UNTIL" expected

The compiler cannot find the reserved word UNTIL to complete the REPEAT statement.

? PAS054 "DO" expected

The compiler cannot find the reserved word DO to complete the FOR statement or the WHILE statement.

? PAS055 "TO" or "DOWNTO" expected

The compiler cannot find the reserved word TO or DOWNTO in the FOR statement.

? PAS058 Invalid expression

The statement syntax requires an expression, but the first symbol the compiler finds is not legal in the expression.

? PAS059 Error in variable

A reference to an array element or record field contains a syntax error.

? PAS060 "ARRAY" expected

The compiler cannot find the reserved word ARRAY in the type definition.

? PAS061 "PROCEDURE" or "FUNCTION" expected

The statement syntax requires the reserved word PROCEDURE or FUNCTION.

? PAS062 Internal compiler error

An internal error has been detected.

? PAS095 Functions "BIN", "OCT", or "HEX" not allowed in this context

The context does not allow the functions BIN, OCT, or HEX.

? PAS096 File component may not exceed 65535 words

The file component is larger than 65535 words.

? PAS097 Error count exceeds error limit. Compilation aborted.

The number of errors exceeded the limit you specified.

? PAS099 End of input encountered before end of program. Compilation aborted.

? PAS100 Array size too large

A declared array is larger than 2,147,483,647 bytes or 2,147,483,647 bits for a packed array.

? PAS101 Identifier declared twice

An identifier is declared twice within a declaration section. You can redeclare identifiers only in different declaration sections.

? PAS102 Lowbound exceeds highbound

The lower limit of a subrange is greater than the upper limit of the subrange, based on their ordinal values in their base type.

? PAS103 Identifier is not of appropriate class

The identifier names the wrong class of data. For example, it names a constant where the syntax of the statement requires a procedure.

? PAS104 Identifier not declared

The program uses an identifier that has not been declared.

? PAS105 Sign not allowed

A plus or minus sign was found in front of an expression of nonnumeric type.

? PAS106 Identifier previously used in this block.

You gave two items the same identifier within the same block.

? PAS107 Incompatible subrange types

The subrange types are not compatible according to the rules of type compatibility.

? PAS108 File not allowed in variant part

A file type cannot appear in the variant part of a record.

? PAS109 Type must not be real or double

You cannot specify a real value here. Real values cannot be used as array subscripts, control values for FOR loops, tag fields of variant records, elements of set expressions, or boundaries of subrange types.

? PAS110 Tagfield type must be scalar or subrange

The tag field for a variant record must be a scalar or subrange type.

? PAS111 Incompatible with tagfield type

The case label and the tag field are of incompatible types. These two items must be compatible according to the general compatibility rules.

? PAS112 Index type must not be real or double

Array subscripts cannot be real values; if numeric, they must be integer or integer subrange values.

? PAS113 Index type must be scalar or subrange

Array subscripts must be scalar or subrange values, and cannot be of a structured type.

? PAS114 Base type must not be real or double

The base type of this set or subrange cannot be one of the real types.

? PAS115 Base type must be scalar or subrange

The base type of this set or subrange must be scalar or subrange values, and cannot be of structured type.

? PAS116 Actual parameter must be a set of correct size

The actual parameter must be of correct size when passed as a VAR parameter.

? PAS117 Undefined forward reference in type declaration: name

Compilation aborted.

? PAS118 Value initialization must be in main program

A VALUE initialization must be in the main program.

? PAS119 Forward declared: repetition of parameter list not allowed

You cannot repeat the parameter list after the forward declaration of a subprogram.

? PAS120 Function result type must be scalar, subrange, or pointer

The function specifies a result that is not a scalar, subrange, or pointer type. Function results cannot be structured types.

? PAS121 File value parameter not allowed

A file cannot be passed as a value parameter.

? PAS122 Forward declared function: repetition of result type not allowed

The result of the function appears in both the forward declaration and in the later complete declaration. The result can appear only in the forward declaration.

? PAS123 Missing result type in function declaration

The function heading does not declare the type of the result of the function.

? PAS124 Fraction format for real and double only

You can specify two integers in the field width (such as R:3:2) for real, single, and double values only.

? PAS125 Error in type of predefined function parameter

A parameter passed to a predefined function is not of the correct type.

? PAS126 Number of parameters does not agree with declaration

The number of actual parameters passed to the subprogram is different from the number of formal parameters declared for that subprogram. You cannot add or omit parameters.

? PAS127 Parameter cannot be element of a packed structure

You cannot pass one element of a packed structure to a subprogram; you must pass the entire structure if you want to use it.

? PAS128 Result type of actual function parameter does not agree with declaration

The result of an actual function parameter is not of the type specified in the formal parameter list.

? PAS129 Operands are of incompatible types

Two or more of the operands in an expression are of incompatible types. For example, the program attempted to compare a numeric and a character variable.

? PAS130 Expression is not of set type

The operators you specified are valid only for set expressions.

? PAS131 Type of variable is not set

The statement syntax requires a set variable.

? PAS132 Strict inclusion not allowed

You must use the <= and >= operators to test set inclusion. PASCAL does not allow you to use the less than (<) and greater than (>) signs for this purpose.

? PAS133 File comparison not allowed

Relational operators cannot be applied to file variables.

? PAS134 Illegal type of operand(s)

You cannot perform the specified operation on data items of the specified types.

? PAS135 Type of operand must be Boolean

This operation requires a Boolean operand.

? PAS136 Set element must be scalar or subrange

The elements of a set must be scalar or subrange types. Sets cannot have elements of structured types.

? PAS137 Set element types not compatible

The elements of this set are not all of the same type.

? PAS138 Type of variable is not an array

A variable that is not an array type is followed by a left square bracket or a comma inside square brackets.

? PAS139 Index type is not compatible with declaration

The specified array subscript is not compatible with the type specified in the array definition.

? PAS140 Type of variable is not record

A period appears following a variable that is not a record type.

? PAS141 Type of variable must be file or pointer

A circumflex character appears after the name of a variable that is not a file pointer.

? PAS142 Illegal parameter substitution

The type of an actual parameter is not compatible with the type of the corresponding formal parameter.

? PAS143 Loop control variable must be an unstructured non-floating point scalar

The control variable in a FOR loop must be an integer, integer subrange, or user-defined scalar type; it cannot be a real variable.

? PAS145 Type conflict between control variable and loop bounds

The type of the control variable in a FOR loop is incompatible with the type of the bounds you specified.

? PAS146 Assignment of files not allowed

You cannot assign one file to another. Output procedures must be used to give values to files.

? PAS147 Label type incompatible with selecting expression

The type of case label is incompatible with the type to which the selecting expression evaluates. Case labels and selecting expressions must be of compatible types.

? PAS148 Subrange bounds must be scalar

You can specify subranges of scalar types only. You cannot specify a real or string subrange.

? PAS149 Index type must not be integer

The index type of a nonconformant array cannot be integer, although it can be an integer subrange.

? PAS150 Assignment to this function is not allowed

You cannot assign a value to an external or predeclared function identifier.

? PAS151 Assignment to formal function parameter not allowed

You cannot assign a value to the name of a formal function parameter.

? PAS152 No such field in this record

You attempted to access a record by an incorrect or nonexistent field name.

? PAS153 Type of parameter must be character string (array of char)

The actual parameter passed to this function or procedure must be a character string.

? PAS154 Type of parameter must be integer

The actual parameter passed to this function or procedure must be an integer.

? PAS155 Recursive %INCLUDE not allowed. Compilation aborted.

? PAS156 Multidefined case label

The same case label refers to more than one statement. Each case label can be used only once within the CASE statement.

? PAS157 Case label range exceeds 1000

The range of ordinal values between the largest and smallest case labels must not exceed 1000.

? PAS158 Missing corresponding variant declaration

In a call to NEW or DISPOSE, more tagfield constants were specified than the number of nested variants in the record type to which the pointer refers.

? PAS159 Double, real or string tagfields not allowed

Tag fields cannot be real or string variables, but must be scalar.

? PAS160 Previous declaration was not forward

The reiteration of a procedure or function that was not forward declared is illegal.

? PAS161 Procedure/function has already been forward declared

The subprogram has already been forward declared.

? PAS162 Undeclared procedure or function: name Compilation aborted.

You specified a procedure or function without declaring it in the declaration section.

? PAS163 Type of parameter must be real or integer

The subprogram requires a real or integer expression as a parameter.

? PAS164 This procedure/function cannot be actual parameter

The specified predeclared procedure or function cannot be an actual parameter. If you must use it in the subprogram, call it directly.

? PAS165 Multiply defined label

A label appears in front of more than one statement in a single executable section.

? PAS166 Multiply declared label

The program declares the same label more than once.

? PAS167 Undeclared label

The program contains a label that has not been declared.

? PAS168 Undefined label: label number Compilation aborted.

You specified a label as an argument to the GOTO statement, but the label is not defined.

? PAS169 Set element value must not exceed 255

The ordinal value of an element of a set must be between 0 and 255.

? PAS170 Value parameter expected

A subprogram that is passed as an actual parameter can have only value parameters.

? PAS171 Type of variable must be textfile (file of char)

The specified operation or subprogram requires a text file variable as an operand or parameter.

? PAS172 Undeclared external file: name Compilation aborted.

You specified an external file that was not declared in the declaration section.

? PAS173 Negative set elements not allowed

The value of an integer set element must be between 0 and 255.

? PAS174 Parameter must be a file type

The specified subprogram requires a file as a parameter.

? PAS175 "INPUT" not declared as an external file

The program makes an implicit reference to the file variable INPUT, but INPUT is either not declared or has been redeclared at an inner level.

? PAS176 "OUTPUT" not declared as an external file

The program makes an implicit reference to the file variable OUTPUT, but OUTPUT is either not declared or has been redeclared at an inner level.

? PAS177 Assignment to function identifier not allowed here

Assignment to a function identifier is allowed only within the function block.

? PAS178 Multidefined record variant

A constant tag field value appears more than once in the definition of a record variant.

? PAS179 File of file type not allowed

You cannot declare a file that has components of a file type.

? PAS181 Array bounds too large

The bounds of an array are too large to allow the elements of the array to be accessed correctly.

? PAS182 Expression must be scalar

The expression must specify a scalar value; structured variables are not legal.

? PAS183 "[GLOBAL]" or "[FORTRAN]" may only precede declarations at level 1

The words [GLOBAL] or [FORTRAN] can be placed only in a function or procedure heading.

? PAS184 External procedure has same name as main program

Program and procedure names must be unique.

? PAS186 Formal procedures may not have conformant array parameters

You cannot pass a conformant array as a parameter to a procedure that is itself passed as a parameter.

? PAS187 Illegal conformant array assignment

The program attempts to perform an illegal assignment involving conformant arrays.

? PAS188 Parameter must be scalar and not real or double

The parameters to the predeclared functions SUCC and PRED must be scalar types, and cannot be one of the real types.

? PAS189 Actual parameter must be a variable

When you use VAR with a formal parameter, the corresponding actual parameter must be a variable and not a general expression.

? PAS190 "READLN", "WRITELN" and "PAGE" are defined only for textfiles

The predeclared procedures READLN, WRITELN, and PAGE operate only on text files.

? PAS191 "READ" and "WRITE" require input/output parameter list

The READ and WRITE procedures require at least one parameter; you cannot omit the parameter list.

? PAS192 Illegal type of input/output parameter

Arrays, sets, records, and pointers cannot be parameters to the READ and WRITE procedures.

? PAS193 Field width parameter must be of type integer

The field width you specify must be an integer.

? PAS194 Variable must be of type PACKED ARRAY[1..11] OF CHAR

The DATE and TIME procedures require a parameter of PACKED ARRAY [1..11] OF CHAR.

? PAS195 Type of variable must be pointer

The statement syntax requires a variable of pointer type.

? PAS196 Type of constant does not agree with tagfield type

The type of a constant in a tag value list is incompatible with the tag field type.

? PAS197 Type of parameter must be real or double

The statement syntax requires a real (single- or double-precision) value.

? PAS198 Type of parameter must be double

The statement syntax requires a double-precision value.

? PAS199 Parameter must be of numeric type

The procedure or function requires an integer or real number value.

? PAS200 Parameter must be scalar or pointer and not real

The procedure or function requires an integer, user-defined scalar, Boolean, integer subrange, user-defined scalar subrange, or pointer parameter.

? PAS201 Error in real constant: digit expected

A real constant contains a nonnumeric character where a numeral is required.

? PAS202 String constant must not exceed source line

The end of the line occurs before the apostrophe that closes a string. Make sure that the second apostrophe has not been left out.

? PAS203 Integer constant exceeds range

An integer constant is outside the permitted range of integers (that is, 2**31 to 2**31).

? PAS204 Actual parameter is not correct type

The actual parameter is not compatible in type with the corresponding formal parameter.

? PAS205 Zero length string not allowed

You cannot specify a string that has no characters.

? PAS206 Illegal digit in binary, octal or hexdecimal constant

A binary, octal, or hexadecimal constant contains an illegal digit.

? PAS207 Real or double constant out of range

A single- or double-precision real number is outside the permitted range -- 0.29*10**(-38) to 1.7*(10**38) for positive numbers and -0.29*10**(-38) to -1.7*(10**38) for negative numbers.

? PAS208 Data type cannot be initialized

This variable contains a type, such as a file, that cannot be initialized.

? PAS209 Variable has been previously initialized

You can specify only one VALUE declaration for a variable.

? PAS210 Variable is not array or record type

The VALUE initialization for a variable that is not a record or an array contains a constructor.

? PAS211 Incorrect number of values for this variable

The VALUE declaration contains too many or too few values for the variable being initialized.

? PAS212 Repetition factor must be positive integer constant

The repetition factor in an array initialization must be a positive integer constant.

? PAS213 Type identifier does not match type of variable

The optional type identifier must be compatible with the type of variable to be initialized.

? PAS214 Incorrect type of value element

A constant appearing in a VALUE initialization has a type other than that of the variable, record field, or array element to be initialized.

? PAS215 RMS record size must be a positive integer constant

The record size specified in the OPEN procedure call was not a positive integer constant.

? PAS216 "OLD" is not allowed for this file

You cannot specify OLD for an internal file.

? PAS217 Assignment to Conformant Array Index is not allowed

You cannot make this assignment to a conformant array index.

? PAS218 Array must be unpacked

An array parameter to PACK or UNPACK is not unpacked correctly.

? PAS219 Array must be packed

An array parameter to PACK or UNPACK is not packed correctly.

? PAS220 Packed bounds must not exceed unpacked bounds

The bounds of the packed array exceed the unpacked bounds.

? PAS224 "[OVERLAID]" expected

The statement syntax requires the keyword [OVERLAID].

? PAS225 Illegal file attribute specification

You specified an attribute in the OPEN statement that is not recognized by the compiler.

? PAS226 Positional parameter not allowed after first non-positional parameter

? PAS227 "OLD", "NEW", "READONLY", or "UNKNOWN" expected

The statement syntax requires either the keyword OLD or NEW.

? PAS228 "SEQUENTIAL" or "DIRECT" expected

The statement syntax requires either the keyword SEQUENTIAL or DIRECT.

? PAS229 "FIXED or "VARIABLE" expected

The statement syntax requires either the keyword FIXED or VARIABLE.

? PAS230 "NOCARRIAGE", "NONE", "CARRIAGE", "FORTRAN", or "LIST" expected

The statement syntax requires one of the following keywords: NOCARRIAGE, NONE, CARRIAGE, FORTRAN, or LIST.

? PAS231 Illegal keyword

This keyword cannot be specified in this context.

? PAS232 Parameter has already been specified

You have specified the same parameter twice.

? PAS233 File variable must be specified

You forgot to specify the file variable.

? PAS234 Identifier or character string literal expected

You need to specify a identifier or character string literal.

? PAS235 Parameter cannot be specified in this position

You specified a parameter that did not belong in this place.

? PAS237 A "NEW" and "DIRECT" file must have fixed-length records

You specified variable-length records for a file that must have fixed-length records.

? PAS238 Record type may not be "VARIABLE" for "DIRECT" files

You cannot have variable length records for a DIRECT file.

? PAS239 %INCLUDE file not found. Compilation aborted.

? PAS240 Include/Exclude file error. Compilation aborted.

? PAS250 Too many nested scopes of identifiers

You can have only 20 levels of nesting. A new nesting level occurs with each block or WITH statement.

? PAS251 Too many nested procedures and/or functions

Subprograms can be nested no more that 20 levels deep.

? PAS252 Assignment to function not allowed here. Probable name scope conflict

A function is nested within a function with the same name.

? PAS253 Too many arguments in an "OPEN" statement

You specified too many arguments.

? PAS255 Too many errors on this source line

The PASCAL compiler diagnoses only the first 20 errors on each source line.

? PAS259 Expression too complicated

The expression is too deeply nested. To correct this error, you should separately evaluate some parts of the expression.

? PAS261 Declarations out of order or repeated declaration sections

The declarations must be in the following order: labels, constants, types, variables, values, and subprograms. Only the main program can contain value declarations.

? PAS264 Only "VAR" parameters allowed for "[FORTRAN]" declared routines

You cannot specify value parameters for [FORTRAN] declared routines.

? PAS265 Parameter not allowed for "[FORTRAN]" declared routines

This type of parameter is not allowed for [FORTRAN] declared routines.

? PAS266 Conformant part of Conformant arrays can have only one dimension packed

You packed more than one dimension in a conformant array part.

? PAS267 Conformant arrays must be of the same type

When using relational operators with conformant arrays, the array types must be equivalent.

? PAS300 Division by zero

The program attempts to divide by zero.

? PAS302 Index expression out of bounds

The value of the expression is out of range for the array element to which you are assigning it.

? PAS303 Value to be assigned is out of bounds

The value to the right of the assignment operator is out of range for the variable to which it is being assigned.

? PAS304 Set element expression out of range

The value of the expression is out of range for the set element to which you are assigning it.

? PAS305 Field width must be greater than zero.

? PAS306 Index type of conformant array parameter exceeds range of declaration

The index type of the actual conformant array parameter extends beyond the range declared in the formal parameter list.

? PAS307 Modulus with zero or negative value

The program tried to take the mod of zero.

? PAS309 Variable space exceeds low segment memory size, 377777 (octal)

? PAS310 Code space exceeds available address space, 777777 (octal)

Compile-Time Warnings

% PAS401 Identifier exceeds, 31, characters

Identifiers can be any length, but PASCAL scans only the first 31 characters for uniqueness.

% PAS402 Error in option specification

A compiler option is incorrectly specified in the source code.

% PAS403 Source input after "END." ignored

The compiler ignores any characters after the END that terminates the program.

% PAS404 Duplicate external procedure name

Two external procedures or functions have been declared with the same name. They refer to the same externally compiled subprogram.

% PAS405 LABEL declaration in MODULE ignored

The compiler ignores label declarations at the outermost level in a module.

% PAS407 Illegal option on include file specification; LIST assumed

You specified an option that is not available.

% PAS408 One or more parameter values assumed before "param"

% PAS409 Alternative ordering of HISTORY and RECORDLENGTH parameters

% PAS410 Parameter type is not known by FORTRAN.

% PAS413 Case label out of range

```
% PAS450  Nonstandard PASCAL:   Exponentiation
% PAS451  Nonstandard PASCAL:   VALUE declaration
% PAS452  Nonstandard PASCAL:   OTHERWISE clause
% PAS453  Nonstandard PASCAL:   %INCLUDE directive
% PAS454  Nonstandard PASCAL:   MODULE declaration
% PAS455  Nonstandard PASCAL:   Label exceeds 9999
% PAS456  Nonstandard PASCAL:   "$" or "" in identifier
```

% PAS457	Nonstandard PASCAL:	Conformant passed to value conformant array
% PAS458	Nonstandard PASCAL:	Directive "[GLOBAL]" or "[FORTRAN]"
% PAS459	Nonstandard PASCAL:	Binary, octal or hexadecimal constant
% PAS460	Nonstandard PASCAL:	Double precision constant
% PAS461	Nonstandard PASCAL:	External procedure declaration
% PAS462	Nonstandard PASCAL:	Binary, octal or hexadecimal data output
% PAS463	Nonstandard PASCAL:	Output of user-defined scalar
% PAS464	Nonstandard PASCAL:	Input of string or user-defined scalar
% PAS465	Nonstandard PASCAL:	Input/output of double precision data
% PAS466	Nonstandard PASCAL:	Implementation-defined type, function, or procedure
% PAS467	Nonstandard PASCAL:	Directive "[OVERLAID]"
% PAS468	Nonstandard PASCAL:	Formal and actual parameters not of identical type
% PAS469	Nonstandard PASCAL:	Control variable is not local
% PAS470	Nonstandard PASCAL:	Formal and actual parameters not both packed or unpacked
% PAS471	Nonstandard PASCAL:	No parameter list declared for this call
% PAS472	Nonstandard PASCAL:	No parameter list declared for this format
% PAS473	Nonstandard PASCAL:	Nonstandard parameter declaration
% PAS474	Nonstandard PASCAL:	Array or record types not identical in assignment
% PAS475	Nonstandard PASCAL:	Types not identical in comparison
% PAS476	Nonstandard PASCAL:	VAR parameter is selector of variant record
% PAS477	Nonstandard PASCAL:	Parameter is pre-defined procedure or function
% PAS478	Nonstandard PASCAL:	NIL used as constant identifier
% PAS479	Nonstandard PASCAL:	Case constants do not cover range of tag-type
% PAS480	Nonstandard PASCAL:	Input/output of conformant string
% PAS481	Nonstandard PASCAL:	Comparison of conformant strings

APPENDIX B

ASCII CHARACTER SET

Table B-1: The ASCII Character Set

ASCII Decimal Number	**Character**	**Meaning**
0	NUL	Null
1	SOH	Start of heading
2	STX	End of text
3	ETX	End of text
4	EOT	End of transmission
5	ENQ	Enquiry
6	ACK	Acknowledgement
7	BEL	Bell
8	BS	Backspace
9	HT	Horizontal tab
10	LF	Line feed
11	VT	Vertical tab
12	FF	Form feed
13	CR	Carriage return
14	SO	Shift out
15	SI	Shift in
16	DLE	Data link escape
17	DC1	Device control 1
18	DC2	Device control 2
19	DC3	Device control 3
20	DC4	Device control 4
21	NAK	Negative acknowledgement
22	SYN	Synchronous idle
23	ETB	End of transmission block
24	CAN	Cancel
25	EM	End of medium
26	SUB	Substitute
27	ESC	Escape
28	FS	File separator
29	GS	Group separator
30	RS	Record separator
31	US	Unit separator
32	SP	Space or blank
33	!	Exclamation mark
34	"	Quotation mark
35	#	Number sign
36	$	Dollar sign
37	%	Percent sign
38	%	Ampersand
39	'	Apostrophe
40	(	Left parenthesis

Table B-1: The ASCII Character Set (Cont.)

ASCII Decimal Number	Character	Meaning
41	)	Right parenthesis
42	*	Asterisk
43	+	Plus sign
44	,	Comma
45	-	Minus sign or hyphen
46	.	Period or decimal point
47	/	Slash
48	0	Zero
49	1	One
50	2	Two
51	3	Three
52	4	Four
53	5	Five
54	6	Six
55	7	Seven
56	8	Eight
57	9	Nine
58	:	Colon
59	;	Semicolon
60	<	Left angle bracket
61	=	Equal sign
62	>	Right angle bracket
63	?	Question mark
64	@	At sign
65	A	Upper case A
66	B	Upper case B
67	C	Upper case C
68	D	Upper case D
69	E	Upper case E
70	F	Upper case F
71	G	upper case G
72	H	upper case H
73	I	Upper case I
74	J	Upper case J
75	K	Upper case K
76	L	Upper case L
77	M	Upper case M
78	N	Upper case N
79	O	Upper case O
80	P	Upper case P
81	Q	Upper case Q
82	R	Upper case R
83	S	Upper case S
84	T	upper case T
85	U	Upper case U
86	V	Upper case V
87	W	Upper case W
88	X	Upper case X
89	Y	Upper case Y
90	Z	Upper case Z
91	[	Left square bracket
92		Back slash
93	]	Right square bracket
94	^ or	Circumflex or up arrow
95	or _	Back arrow or underscore

Table B-1: The ASCII Character Set (Cont.)

ASCII Decimal Number	Character	Meaning
96	'	Grave accent
97	a	Lower case a
98	b	Lower case b
99	c	Lower case c
100	d	Lower case d
101	e	Lower case e
102	f	Lower case f
103	g	Lower case g
104	h	Lower case h
105	i	Lower case i
106	j	Lower case j
107	k	Lower case k
108	l	Lower case l
109	m	Lower case m
110	n	Lower case n
111	o	Lower case o
112	p	Lower case p
113	q	Lower case q
114	r	Lower case r
115	s	Lower case s
116	t	Lower case t
117	u	Lower case u
118	v	Lower case v
119	w	Lower case w
120	x	Lower case x
121	y	Lower case y
122	z	Lower case z
123	{	Left brace
124	|	Vertical line
125	}	Right brace
126	~	Tilde
127	DEL	Delete

APPENDIX C

SYNTAX SUMMARY

This appendix summarizes the syntax of the PASCAL-20 language in the Backus-Naur Form (BNF):

C.1 BACKUS-NAUR FORM

In the BNF, each element of the language is defined recursively in terms of simpler elements. The element being defined is written to the left of the symbol ::= and its definition is written to the right of that symbol.

The BNF uses a group of metasymbols that differ from the conventions used in the rest of this manual and are not part of the PASCAL language. Table C-1 lists the meta-symbols used in the BNF.

Table C-1: BNF MetaSymbols

Symbol	Meaning
::=	Separates the element being defined from its definition.
< >	Encloses a definable language element.
[ ]	Encloses an optional element.
|	Means "or"; separates possible elements.
{ }	Encloses elements that may be repeated one or more times, but need not be present

The remainder of this section lists PASCAL in BNF.

```
<compilation unit> ::= <program> | <module>

<module> ::= <module heading> <global declaration part>
          <procedure and function declaration part> END .

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

<module heading> ::= <module word> <identifier> ; |
           <module word> <identifier> ( <program parameters> ) ;
```

```
<global declaration part> ::= <label declaration part>
                  <constant definition part>
                  <type definition part>
                  <variable declaration part>

<module word> ::= MODULE | [[OVERLAID]] MODULE

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

<program parameters> ::= <external file identifier>
           { , <external file identifier> }

<external file identifier> ::= <identifier>

<program word> ::= PROGRAM | [[OVERLAID]] PROGRAM

<identifier> ::= <identifier head> {<letter or digit> | _ | $}

<identifier head> ::= <letter> | _ | $

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

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

<special symbol> ::= <operator symbol> | <reserved word> |
           <semireserved word> | <include symbol>

<operator symbol> ::= + | - | * | / | = | <> | < | > | <= | >=
                  | ( | ) | [ | ] | { | } | := | . | , | ; | : | '
                  | ^ | ** | .. | @ | (* | *) | (. | .)

<reserved word> ::= DIV | MOD | NIL | IN | OR | AND | NOT | IF
                  | THEN | ELSE | CASE | OF | REPEAT | UNTIL | WHILE
                  | DO | FOR | TO | DOWNTO | BEGIN | END | WITH | GOTO
                  | CONST | VAR | TYPE | ARRAY | RECORD | SET | FILE
                  | FUNCTION | PROCEDURE | LABEL | PACKED | PROGRAM

<semireserved word> ::= REM | OTHERWISE | MODULE | VALUE

<include symbol> ::= %INCLUDE

<blank> ::= <single blank> { <single blank> }

<single blank> ::= <space character> | <tab character>

<block> ::= <declaration part> <value initialization part>
            <procedure and function declaration part> <statement part>

<declaration part> ::= <label declaration part> <constant declaration
           part> <type declaration part> <variable declaration part>

<label declaration part> ::= <empty> | LABEL <label> { , <label> } ;

<label> ::= <unsigned integer>

<constant definition part> ::= <empty> |
            CONST <constant definition> { ; <constant definition> } ;

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

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

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

<unsigned integer> ::= <digit sequence> | <radix integer>

<radix integer> ::= % <octal integer> | % <hex integer> |
                    % <binary integer>

<octal integer> ::= <letter o> ' <octal digit sequence> '

<letter o> ::= O | o

<octal digit sequence> ::= <octal digit> { <octal digit> }

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

<hex integer> ::= <letter x> ' <hex digit sequence> '

<letter x> ::= X | x

<hex digit sequence> ::= <hex digit> { <hex digit> }

<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d
                        | e | f

<binary integer> ::= <letter b> ' <binary digit sequence> '

<letter b> ::= B | b

<binary digit sequence> ::= <binary digit> { <binary digit> }

<binary digit> ::= 0 | 1

<unsigned real> ::= <unsigned single> | <unsigned double>

<unsigned single> ::= <digit sequence> . <digit sequence> |
           <digit sequence> . <digit sequence> E <scale factor> |
           <digit sequence> . <digit sequence> e <scale factor> |
           <digit sequence> E <scale factor> |
           <digit sequence> e <scale factor>

<unsigned double> ::= <digit sequence> . <digit sequence> D <scale
           factor> | <digit sequence> . <digit sequence> d <scale
           factor> | <digit sequence> D <scale factor> |
           <digit sequence> d <scale factor>

<digit sequence> ::= <digit> { <digit> }

<scale factor> ::= <digit sequence> | <sign> <digit sequence>

<sign> ::= + | -

<constant identifier> ::= <identifier>

<string> ::= ' <character> { <character> } '

<character> ::= <any ASCII character except '> | ''

<type definition part> ::= <empty> |
           TYPE <type definition> { ; <type definition> } ;
```

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

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

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

<scalar type> ::= ( <identifier> { , <identifier> } )

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

<type identifier> ::= <identifier>

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

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

<array type> ::= ARRAY [ <index type> { , <index type> } ] OF
          <component type>

<index type> ::= <simple type>

<component type> ::= <type>

<record type> ::= RECORD <field list> END

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

<fixed part> ::= <record section>{ ; <record section> }

<record section> ::= <empty> |
          <field identifier> { , <field identifier> } : <type>

<variant part> ::= CASE <tag field> <type identifier> OF
          <variant> { ; <variant> }

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

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

<case label list> ::= <case label> { , <case label> }

<case label> ::= <constant>

<set type> ::= SET OF <base type>

<base type> ::= <simple type>

<file type> ::= FILE OF <type>

<pointer type> ::= ^ <type identifier>

<variable declaration part> ::= <empty> |
          VAR <variable declaration> { ; <variable declaration> } ;

<variable declaration> ::= <identifier list> : <type>
            | <identifier list> : <type> := <value>

<identifier list> ::= <identifier> { , <identifier> }

<value initialization part> ::= VALUE <value initialization>
                        { ; <value initialization> } ; | <empty>
```

```
<value initialization> ::= <identifier> := <value>

<value> ::= <constant> | <set constant> | <constructor>

<set constant> ::= [ <constant element list> ]

<constant element list> ::= <empty> |
                       <constant element> { , <constant element> }

<constant element> ::= <constant> | <constant> .. <constant>

<constructor> ::= <optional type> ( <value element>
                                  { , <value element> } )

<optional type> ::= <empty> | <type identifier>

<value element> ::= <value> | <repetition factor> OF <value>

<repetition factor> ::= <unsigned integer> | <constant identifier>

<procedure or function declaration part> ::=
           { <procedure or function declaration> ; }

<procedure or function declaration> ::= <procedure declaration> |
           <function declaration>

<procedure declaration> ::= <internal procedure declaration> |
           <external procedure declaration> |
           <forward procedure declaration>

<internal procedure declaration> ::= <procedure heading> <block> |
                           [[GLOBAL]] <procedure heading> <block> |
                           [[FORTRAN]] <procedure heading> <block>

<external procedure declaration> ::= <procedure heading> EXTERN |
          <procedure heading> EXTERNAL | <procedure heading> FORTRAN

<forward procedure declaration> ::= <procedure heading> FORWARD
                           [[GLOBAL]] <procedure heading> FORWARD |
                           [[FORTRAN]] <procedure heading> FORWARD

<procedure heading> ::= PROCEDURE <identifier> ; |
           PROCEDURE <identifier> ( <formal parameter section>
           { ; <formal parameter section> } ) ;

<formal parameter section> ::= <extended parameter group> |
           VAR <extended parameter group> |
           FUNCTION <parameter group> |
           PROCEDURE <identifier> { , <identifier> } |
           <procedure heading> | <function heading>

<extended parameter group> ::= <parameter group> |
           <identifier> { , <identifier> } : <conformant array schema>

<conformant array schema> ::= ARRAY [ <index type specification>
           { ; <index type specification> } ] OF <type identifier> |
           ARRAY [ <index type specification>
           { , <index type specification> } ] OF <conformant array
           schema> PACKED ARRAY [ <index type specification> ]
           OF <type identifier>

<index type specification> ::= <identifier> .. <identifier> :
                          <scalar type identifier>
```

```
<scalar type identifier> ::= <identifier>

<parameter group> ::= <identifier> { , <identifier> } :
                               <type identifier>

<function declaration> ::= <internal function declaration> |
                           <external function declaration> |
                           <forward function declaration>

<internal function declaration> ::= <function heading> <block> |
                           [GLOBAL] <function heading> <block> |
                           [FORTRAN] <function heading> <block>

<external function declaration> ::= <function heading> EXTERN |
          <function heading> EXTERNAL | <function heading> FORTRAN

<forward function declaration> ::= <function heading> FORWARD |
                           [GLOBAL] <function heading> FORWARD |
                           [FORTRAN] <function heading> FORWARD

<function heading> ::= FUNCTION <identifier> : <result type> ; |
                  FUNCTION <identifier> ( formal parameter section>
                  { ; <formal parameter section> } ) : <result type> ;

<result type> ::= <type identifier>

<statement part> ::= <compound statement>

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

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

<simple statement> ::= <assignment statement> | <procedure statement> |
                           <go to statement> | <empty statement>

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

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

<entire variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

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

<indexed variable> ::= <array variable> [ <expression>
                           { , <expression> } ]

<array variable> ::= <variable>

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

<record variable> ::= <variable>

<field identifier> ::= <identifier>

<file buffer> ::= <file variable> ^

<file variable> ::= <variable>
```

```
<referenced variable> ::= <pointer variable> ^

<pointer variable> ::= <variable>

<expression> ::= <simple expression> | <simple expression>
                          <relational operator> <simple expression>

<relational operator> ::= = | <> | < | <= | >= | > | IN

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

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

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

<multiplying operator> ::= * | / | DIV | MOD | AND | REM

<primary> ::= <factor> | <primary> ** <factor>

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

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

<function designator> ::= <function identifier> |
                      <function identifier> ( <actual parameter>
                      { , <actual parameter> } )

<function identifier> ::= <identifier>

<set> ::= [ <element list> ]

<element list> ::= <element> { , <element> } | <empty>

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

<procedure statement> ::= <procedure identifier> |
                      <procedure identifier> ( <actual parameter>
                      { , <actual parameter> } )

<procedure identifier> ::= <identifier>

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

<go to statement> ::= GOTO <label>

<empty statement> ::= <empty>

<empty> ::=

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

<compound statement> ::= BEGIN <statement> { ; <statement> } END

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

<if statement> ::= IF <expression> THEN <statement> |
                 IF <expression> THEN <statement> ELSE <statement>
```

```
<case statement> ::= CASE <expression> OF <case list element>
              { ; <case list element> } <otherwise part> <end case>

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

<otherwise part> ::= <empty> |
                          OTHERWISE <statement> { ; <statement> } |
                          ; OTHERWISE <statement> { ; <statement> }

<end case> ::= END | ; END

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

<while statement> ::= WHILE <expression> DO <statement>

<repeat statement> ::= REPEAT <statement> { ; <statement> }
                            UNTIL <expression>

<for statement> ::= FOR <control variable> := <for list> DO <statement>

<for list> ::= <initial value> TO <final value> |
                            <initial value> DOWNTO <final value>

<control variable> ::= <identifier>

<initial value> ::= <expression>

<final value> ::= <expression>

<with statement> ::= WITH <record variable list> DO <statement>

<record variable list> ::= <record variable> { , <record variable> }

<open statement> ::= OPEN ( <open parameters> )

<open parameters> ::= <file parameters>
                 | <keyword parameters> { , <keyword parameters> }

<keyword parameters> ::= FILE_VARIABLE := <file variable>
                            | FILE_NAME := <file name>
                            | HISTORY := <file status>
                            | RECORD_LENGTH := <record length>
                            | ACCESS_METHOD := <record access mode>
                            | RECORD_TYPE := <record type>
                            | CARRIAGE_CONTROL := <carriage control>

<file name> ::= <variable> | <constant identifier>
                            | ' <file specification> '

<file status> ::= OLD | NEW | UNKNOWN | READONLY

<record length> ::= <unsigned integer>

<record access mode> ::= SEQUENTIAL | DIRECT

<record type> ::= VARIABLE | FIXED

<carriage control> ::= CARRIAGE | NOCARRIAGE | LIST
```

```
<file parameters> ::= <file variable> <RMS file specification>
                          <RMS file history> <RMS record length>
                          <RMS record access mode>
                          <RMS record type>
                          <RMS carriage control>

<RMS file specification> ::= , ' <file specification> ' | <empty>

<RMS record length> ::= , <record length> | <empty>

<RMS file history> ::= , <file status> | <empty>

<RMS record access mode> ::= , <record access mode> | <empty>

<RMS record type> ::= , <record type> | <empty>
```

APPENDIX D

SUMMARY OF PASCAL-20 EXTENSIONS TO PROPOSED ISO STANDARD

Category	Extension
Lexical and syntactical	Semireserved words: MODULE, OTHERWISE, REM
	Exponentiation operator (**)
	REM operator
	Binary, hexadecimal, and octal notation for integers
	Double-precision real data type
	Dollar sign ($) and underscore (_) characters in identifiers
	Identifiers of up to 31 characters
Predefined types	SINGLE, DOUBLE
Predeclared Procedures	CLOSE, DATE, FIND, HALT, LINELIMIT, OPEN, TIME, MARK, RELEASE
READ, READLN, WRITE, and WRITELN extensions	Parameters of character string and enumerated types for READ and READLN
	Parameters of enumerated types for WRITE and WRITELN
READ, READLN, WRITE, and WRITELN extensions	Optional carriage-control specification for text files with WRITE and WRITELN
Declarations	Variable initialization
Statements	OTHERWISE clause in CASE statement
Procedures and Functions	External procedure and function declaration
	Support for calling externally declared FORTRAN subroutines and for declaration of PASCAL subroutines that can be called by FORTRAN
Compilation	MODULE capability for combining declarations and definitions to be compiled independently from the main program

APPENDIX E

ISO COMPLIANCE

This appendix is a statement of the compliance of PASCAL-20 with the PASCAL standard ISO 7185. It is divided into four sections:

1. Implementation-defined features. This is a list of the features of PASCAL that must be defined by each implementation, but which may vary between implementations.

2. Implementation-dependent features. This is a list of the features of PASCAL which may or may not be defined by an implementation.

3. Errors. This is an explanation of how PASCAL-20 handles each of the errors defined by ISO 7185.

4. Exceptions and restrictions. This is a list of violations of the standard which PASCAL-20 does not detect, and restrictions imposed by PASCAL-20.

This appendix does not list extensions to the ISO 7185 standard implemented by PASCAL-20. These extensions are listed in Appendix D.

E.1 IMPLEMENTATION-DEFINED FEATURES

The ISO 7185 standard leaves the exact definition of a number of language elements up to the implementation. Following is a list of those features, and the definition given them by PASCAL-20.

1. The predefined type CHAR corresponds to the 7-bit ASCII character set.

2. The identifier MAXINT denotes a value of 2**35-1, which is the maximum integer value of a PDP-10 word.

3. In a WRITE or WRITELN statement, the default field-width for integers is 10, for real numbers is 16, and for Booleans is 16. The number of digits written in the exponent of a floating-point format real is 2. On output to a text file, the Boolean values TRUE and FALSE are translated to the uppercase character strings "TRUE" and "FALSE" respectively.

4. A program parameter that is a file variable is bound at runtime to an external (operating system) file. An external file may exist prior to execution of the program and is not deleted when the program exits. PASCAL-20 determines which file specification to use when opening the file in the following manner:

   - If an OPEN statement is executed prior to a RESET or REWRITE of the file, the file specification given in the OPEN statement is used.

   - If no OPEN statement is executed, the RESET or REWRITE looks for a logical name with the same name as the file variable. If it finds the logical name, that name is used as the file specification.

   - Otherwise, the variable name is used as the file name, and an extension of .DAT is assumed.

The ISO 7185 standard also states that the precise actions of the I/O procedures REWRITE, PUT, RESET, GET, and PAGE are implementation defined. In the following items, F is assumed to be a file variable, and the various pre- and post assertions for each operation are assumed to be true.

1. REWRITE(F) creates or supersedes an operating system file, and opens the file for writing.

2. PUT(F), where F is a text file, writes the contents of the file buffer variable to the output buffer, and advances the buffer pointer; no file operation is performed.

3. PUT(F), where F is not a text file, writes the contents of the file buffer variable to the file.

4. RESET(F) opens an existing operating system file for reading, reads the first record, and assigns a value to the file buffer variable.

5. GET(F), where F is a text file, advances the file input buffer pointer. If EOLN(F) is true, the next record is read from the file, and the buffer pointer is reset to the beginning of the buffer.

6. GET(F), where F is not a text file, reads the next record from the file.

7. PAGE(F) first flushes the output buffer to the file. Then, it causes the file to skip to the top of the next page; the mechanics of this depend on the setting of the file's carriage-control attribute. If this attribute is LIST (the default), a form feed character (control-L) is written to the file. If the attribute is CARRIAGE or FORTRAN, a '1' is written to the file. If the attribute is NOCARRIAGE or NONE, an error is generated.

E.2 IMPLEMENTATION-DEPENDENT FEATURES

These language elements are defined by the ISO 7185 standard as "possibly differing between processors and not necessarily defined for any particular processor." Following is a description of how PASCAL-20 treats each of these elements.

1. The order of evaluation of array indices is their lexical order (that is, left to right).

2. The order of evaluation of members of a set constructor is their lexical order (that is, left to right).

3. The order of evaluation of the operands of a dyadic operator (for example, '+') depends on the complexity of each operand. Both operands are always evaluated (even in Boolean expressions which could be "short-circuited").

4. The order of evaluation, accessing, and binding of actual parameters to formal parameters in a procedure or function call is their lexical order (that is, left to right).

5. The variable in an assignment statement is accessed after the evaluation of the expression.

6. The effect of reading a file at the point where a PAGE was executed when writing the file is to return either a form-feed character (control-L) or the digit '1', depending on the status of the file when the PAGE was executed. (See above for a description of PAGE's actions.)

7. Program parameters which are not file variables have no defined meaning in PASCAL-20, and cause an error.

E.3 ERROR HANDLING

This section describes how the PASCAL-20 compiler and run-time system detect violations of level 1 of the standard proposed by the ISO for the PASCAL language. Errors detected at run time cause a program to terminate and return appropriate error messages. Errors described here as "not detected" cause a program to produce unexpected results.

The type of an index value is not assignment compatible with the index type of an array.

Explanation: Detected at run time if checking was enabled during compilation.

The current variant changes while a reference to it exists.

Explanation: Not detected. An example of a reference to a variant is the passing of the variant to a formal VAR parameter.

The value of a variable to which a pointer refers (p) is NIL.

Explanation: Detected if checking was enabled at run time.

The value of a variable to which a pointer refers (p) is undefined.

Explanation: Not detected.

The DISPOSE procedure is called to dispose of a heap-allocated variable while a reference to the variable exists.

Explanation: Not detected. Examples of such references are: passing the variable or a component of it to a formal VAR parameter, or using the variable in a WITH statement (if the variable is a record).

The value of file f changes while a reference to f^ exists.

Explanation: Not detected. An example of a reference to f^ is the passing of f^ by reference to a routine; until the routine has ceased execution, you may not perform any operation on file f.

The ordinal type of an actual parameter is not assignment compatible with the type of the corresponding formal parameter.

Explanation: Detected at run time if checking was enabled during compilation of the called routine.

The set type of an actual parameter is not assignment compatible with the type of the corresponding formal parameter.

Explanation: Detected at run time if checking was enabled during compilation of the called routine.

A file is not in Generation mode when a PUT, WRITE, WRITELN, or PAGE procedure is attempted.

Explanation: Detected at run time.

A file is in Undefined mode when a PUT, WRITE, WRITELN, or PAGE procedure is attempted.

Explanation: Detected at run time.

The result of an EOF function is not TRUE when a PUT, WRITE, WRITELN, or PAGE procedure is attempted.

Explanation: Detected at run time.

The value of the file buffer variable is undefined when a PUT procedure is attempted.

Explanation: Not detected.

A file is in Undefined mode when a RESET procedure is attempted.

Explanation: Not detected.

A file is not in Inspection mode when a GET, READ, or READLN procedure is attempted.

Explanation: Detected at run time.

A file is in Undefined mode when a GET, READ, or READLN procedure is attempted.

Explanation: Detected at run time.

The result of an EOF function is TRUE when a GET, READ, or READLN procedure is attempted.

Explanation: Detected at run time.

The type of the file buffer is not assignment compatible with the type of the variable that is a parameter to a READ or READLN procedure.

Explanation: Detected at run time if checking is enabled during compilation at the READ or READLN.

The type of the expression being written by a WRITE or WRITELN procedure is not assignment compatible with the type of the file buffer variable.

Explanation: Detected at run time if checking is enabled during compilation of the WRITE or WRITELN.

The current variant does not exist in the list of variants specified with the NEW procedure.

Explanation: Not detected.

The DISPOSE(p) procedure is called to deallocate a pointer variable that was created using the variant form of the NEW procedure.

Explanation: Not detected.

The variant form of the DISPOSE procedure does not specify the disposal of the same number of variants that were created by the variant form of the NEW procedure.

Explanation: Not detected.

The variant form of the DISPOSE procedure does not specify the disposal of the same variants that were created by the variant form of the NEW procedure.

Explanation: Not detected.

The value of the parameter to the DISPOSE procedure is NIL.

Explanation: Detected at run time.

The value of the parameter to the DISPOSE procedure is undefined.

Explanation: Usually detected at run time.

A variant record created by the NEW procedure is accessed as a whole, rather than one component at a time.

Explanation: Not detected.

In the PACK(a,i,z) procedure, the type of the index value i is not assignment compatible with the index of a.

Explanation: Not detected.

The PACK procedure is attempted when the value of at least one component of a is undefined.

Explanation: Not detected.

The index value i in the PACK procedure is greater than the upper bound of the index of a.

Explanation: Not detected.

In the UNPACK(z,i,a) procedure, the type of the index value i is not assignment compatible with the index type of a.

Explanation: Not detected.

The index value i in the UNPACK procedure is greater than the upper bound of the index type of a.

Explanation: Not detected.

The UNPACK procedure is attempted when the value of at least one component of z is undefined.

Explanation: Not detected.

The resulting value of SQR(X) does not exist.

Explanation: Detected at run time.

In the expression LN(X), the value of X is negative.

Explanation: Detected at run time.

In the expression, SQRT(X), the value of X is negative.

Explanation: Detected at run time.

The resulting value of TRUNC(X) does not exist after the following calculations have been done: if the value of X is positive or zero, then 0 <= X-TRUNC (X) <1; otherwise, -1 < X-TRUNC (X) <= 0.

Explanation: Detected at run time.

The resulting value of ROUND(X) does not exist after the following calculations have been done: if the value of X is positive or zero, ROUND(X) is equivalent to TRUNC(X+0.5); otherwise, ROUND(X) is equivalent to TRUNC(X-0.5).

Explanation: Detected at run time.

The resulting value of CHR(X) does not exist.

Explanation: Not detected.

The resulting value of SUCC(X) does not exist.

Explanation: Detected at run time if checking was enabled during compilation.

The resulting value of PRED(X) does not exist.

Explanation: Detected at run time if checking was enabled during compilation.

The function EOF(f) is called when the file f is undefined.

Explanation: Detected at run time.

The function EOLN(f) is called when the file f is undefined.

Explanation: Detected at run time.

The function EOLN(f) is called when the result of EOF(f) is TRUE.

Explanation: Detected at run time.

A variable is not initialized before it is first used.

Explanation: Not detected.

In the expression X/Y, the value of Y is zero.

Explanation: Detected at run time.

In the expression I DIV J, the value of J is zero.

Explanation: Detected at run time.

In the expression I MOD J, the value of J is zero or negative.

Explanation: Detected at run time if J is zero; not detected if J is negative.

An operation or function involving integers does not conform to the mathematical rules for integer arithmetic.

Explanation: Detected at run time.

A function result is undefined when the function returns control to the calling block.

Explanation: Not detected.

The ordinal type of an expression is not assignment compatible with the type of the variable or function identifier to which it is assigned.

Explanation: Detected at run time if checking was enabled during compilation.

The set type of an expression is not assignment compatible with the type of the variable or function identifier to which it is assigned.

Explanation: Detected at run time if checking was enabled during compilation.

None of the case labels is equal in value to the case selector in a CASE statement.

Explanation: Not detected.

In a FOR statement, the type of the initial value is not assignment compatible with the type of the control variable, and the statement in the loop body is executed.

Explanation: Detected at run time if checking was enabled during compilation. Assignment compatibility is not enforced if the statement in the loop body can never be executed.

In a FOR statement, the type of the final value is not assignment compatible with the type of the control variable and the statement in the loop body is executed.

Explanation: Detected at run time if checking was enabled during compilation. Assignment compatibility is not enforced if the statement in the loop body can never be executed.

When an integer is being read from a text file, the digits read do not constitute a valid integer value. (Initial spaces and end-of-line markers are skipped.)

Explanation: Detected at run time.

When an integer is being read from a text file, the type of the value read is not assignment compatible with the type of the variable.

Explanation: Detected at run time.

When a real number is read from a text file, the digits read do not constitute a valid real number. (Initial spaces and end-of-line markers are skipped.)

Explanation: Detected at run time.

The value of the file buffer variable is undefined when a READ or READLN procedure is performed.

Explanation: Not detected.

A WRITE or WRITELN procedure specifies a field width in which the integers representing the total width and the number of fractional digits are less than 1.

Explanation: Detected at run time.

The bounds of an array passed to a conformant array parameter are outside the range specified by the conformant array's index type.

Explanation: Not detected.

E.4 EXCEPTIONS AND RESTRICTIONS

PASCAL-20 is not able to detect a number of violations of the ISO 7185 standard. Violations that are not defined as errors are listed below.

1. Statements threatening a FOR-loop control variable are not detected. A statement "threatens" a variable if it occurs within the FOR-loop or within a prior procedure, and if it is one of the following kinds of statements:

   - A READ or READLN statement containing the variable
   - An assignment statement with the variable to the left of the ':='
   - A procedure or function call with the variable as a formal VAR parameter

2. The concept of a "totally-undefined" variable has no meaning in PASCAL-20. Any violation of the standard which requires "totally-undefined" to be meaningful is not detected.

3. Variant records in PASCAL-20 are implemented by overlaying the fields of the variants in the same memory. Changing the active variant does not alter the contents of that memory, and the fields of the newly-active variant are not undefined. Also, note that variant parts of records cannot be of the type FILE.

4. The DISPOSE procedure does not cause its argument to become undefined, but rather the argument is set to NIL.

5. No restrictions on the relative placement of labels and GOTO statements are enforced.

6. Packed and unpacked sets are implemented in similar ways in PASCAL-20. Therefore, the compiler is very "loose" about interchanging packed and unpacked sets; the only requirement imposed on compatibility of set types is that their base types must be compatible.

7. Under certain conditions, a procedure call may be bound to the wrong procedure definition. This can only happen when two procedures are declared with the same name at different levels, and the inner procedure is called before it is declared. The following example illustrates this situation.

```
PROCEDURE B;
BEGIN (* level 1 *)
END;
PROCEDURE C;
   PROCEDURE D;
   BEGIN
   B      (* call bound to level 1 procedure *)
   END;
   PROCEDURE B;
   BEGIN (* level 2 *)
   END;
BEGIN
END;
```

PASCAL-20 also imposes a number of restrictions on programs. These are:

1. Identifiers may be of any length, but only the first 31 characters are stored. Therefore identifiers in PASCAL-20 must be unique in the first 31 characters.

2. The ordinal values of the base type of a set type are limited to the range 0..255. This means that no set may have more than 256 elements.

3. File variables are not permitted as fields in the variant part of a record.

4. The range of ordinal values of case constants in a CASE statement must be less than 1000.

APPENDIX F

DIFFERENCES BETWEEN PASCAL-20 AND VAX-11 PASCAL

PASCAL-20 V1 and VAX-11 PASCAL V2 are compatible languages; that is, you can compile and run a PASCAL-20 program on a VAX/VMS operating system, and you can compile and run a VAX-11 PASCAL program on a TOPS-20 operating system. However, there are differences between the languages.

The VAX-11 PASCAL language (V2) contains features that the PASCAL-20 language does not. Therefore, if you plan to transport PASCAL programs from the VAX/VMS operating system to the TOPS-20 operating system, you should avoid using these additional features. This appendix lists the additional features that VAX-11 PASCAL (V2) has over PASCAL-20 (V1).

Table F-1 lists the language elements that VAX-11 PASCAL V2 has that PASCAL-20 does not:

Table F-1: Additional Language Elements

Category	Language Element	
Special Symbol	Type cast operator	::
Nonstandard Reserved words	%DESCR	
	%IMMED	
	%STDESCR	
	%REF	
	VARYING	
Predeclared Identifiers	ADD_INTERLOCKED	REVERT
	ADDRESS	SET_INTERLOCKED
	BITNEXT	SIZE
	BITSIZE	STATUS
	CLEAR_INTERLOCKED	SUBSTR
	DBLE	TRUNCATE
	DELETE	UAND
	ESTABLISH	UFB
	FINDK	UINT
	INDEX	UNLOCK
	INT	UNOT
	LENGTH	UNSIGNED
	LOCATE	UOR
	LOWER	UPDATE
	NEXT	UPPER
	PAD	GROUND
	QUAD	UNTRUNC
	QUADTRUPLE	UXOR
	READY	WRITEV
	RESETK	
Data Types	UNSIGNED: 0 through 2**32 - 1	
	0 through 4294967295	
Formal Parameter List	foreign section	
Structured Type	VARYING OF CHAR	
Real Types	G_floating	
	QUADRUPLE	
String Operators	Plus sign (+)	
Predeclared Procedures	DELETE(f,e)	
	ESTABLISH(id)	
	FINDK(f,kn,kv,m,e)	
	LOCATE(f,n,e)	
	READV(s,vl,...,vn)	
	RESTK(f,kn,e)	
	REVERT	
	TRUNCATE(f,e)	
	UNLOCK(f,e)	
	UPDATE(f,e)	
	WRITEV(s,pl,...,pn)	

Table F-1: Additional Language Elements (Cont.)

Category		Language Element
Predeclared Functions		See Table F-2
OPEN Parameters	Procedure	DISPOSITION SHARING USER_ACTION ERROR KEYED
CLOSE Parameters	Procedure	DISPOSITION USER_ACTION ERROR

The following predeclared procedures are contained in both PASCAL languages, however, VAX-11 PASCAL offers an additional argument:

```
     e = error parameter
```

These predeclared procedures are:

```
     FIND(f,n,e)
     OPEN(f,parameters,e)
     PAGE(f,e)
     PUT(f,e)
     READ(f,vl,...,vn,e)
     READLN(f,vl,...,vn,e)
     RESET(f,e)
     REWRITE(f,e)
     WRITE(f,pl,...,pn,e)
```

Whenever you see [[attribute-list]] in a syntax description in the VAX-11 PASCAL language, it signifies that you can add certain arguments to the syntax depending on the type of circumstances. Note the following syntax and attribute descriptions:

Attribute-list

```
     [{identifier ⟦ ( { constant-expression },...) ⟧ },...]
```

In the PASCAL-20 language, only a PROGRAM, MODULE, PROCEDURE, or FUNCTION can have an attribute-list. The only attribute for a PROGRAM or MODULE is OVERLAID. The only attributes for a PROCEDURE or FUNCTION are GLOBAL and FORTRAN. Note that in PASCAL-20 the GLOBAL attribute can not be followed by an identifier.

Table F-2 lists additional predeclared functions that VAX-11 PASCAL provides but PASCAL-20 does not:

Table F-2: Additional Predeclared Functions

Category	Function
Boolean	UFB(f)

Transfer	DBLE(x)
	QUAD(x)
	UINT(x)
	UROUND(r)
	UTRUNC(r)

Dynamic Allocation	ADDRESS(x)

Character String	INDEX(s1,s2)
	LENGTH(s)
	PAD(s,fill,l)
	SUBSTR(s,b,l)

Unsigned	UNAD(u1,u2)
	UNOT(u)
	UOR(u1,u2)
	UXOR(u1,u2)

Allocation Size	SIZE(x,c1,...,cn)
	NEXT(x)
	BITSIZE(x)
	BITNEXT(x)

Low_Level Interlocked	ADD_INTERLOCKED(e,v)
	SET_INTERLOCKED(b)
	CLEAR_INTERLOCKED(b)

Miscellaneous	STATUS(f)

If you plan to transport VAX-11 PASCAL programs to TOPS-20, or PASCAL-20 programs to VAX/VMS, note the following:

- The precision for INTEGER data type:

TOPS-20	(-2**35) through (+2**35)-1 -34359738368 through +34359738367
VAX/VMS	(-2**31)+1 through (+2**31)-1 -2147483647 through +2147483647

- The maximum number of items in an enumerated type:

TOPS-20	Depends on amount of memory available
VAX/VMS	65,535

- PASCAL-20 has two procedures MARK and RELEASE that are used in conjunction with NEW and DISPOSE. VAX-11 PASCAL does not have MARK and RELEASE.

If you have any further questions concerning VAX-11 PASCAL, refer to the VAX-11 PASCAL documentation set.

APPENDIX G

PROCEDURE AND FUNCTION CALLING SEQUENCES

This appendix[1] describes the calling sequences and conventions used by PASCAL for user-defined procedures and functions. This information is particularly useful in writing MACRO routines that interface with PASCAL programs.

Note that, in this appendix, the word "procedure" refers to both procedures and functions. Exceptions in the case of functions are explicitly noted.

G.1 RUN-TIME STACK

The majority of run-time information is kept on the stack; in particular, the stack contains all local variables, parameters, static and dynamic links, and return addresses for procedure calls. The stack pointer is kept in AC 17. A frame pointer, used for most variable accessing, is kept in AC 16. The frame pointer is fixed over the course of a procedure, while the stack pointer can change. Figure G-1 shows the status of the stack just after a PUSHJ to a procedure.

```
            ==========================
            |/////////////////////////|
AC 0 ->     | previous stack frame    |
            |/////////////////////////|    <--
         ================================    |
            | parameter values,       |      |
            | addresses, descriptors  |      |
            ==========================       |
            | saved stack pointer     |    >--
            ==========================
AC17 ->     | caller's return address|
            ==========================
```

Figure G-1: Status of Stack After PUSHJ

[1] The information in this appendix is copyrighted by Scott Arthur Moody @ 1982.

Figure G-2 shows the stack frame in its entirety, just prior to the first executable statement in a procedure.

```
          ==========================
          |////////////////////////|
          | previous stack frame   |   <-----
          |////////////////////////|   <--  |
        ==============================    | |
          | parameter values,      |      | |
          | addresses, descriptors |      | |
          ==========================      | |
          | saved stack pointer    |   >--  |
          ==========================        |
          | caller's return address|        |
          ==========================        |
AC16 ->   | dynamic link           |   >-----
          ==========================
          | static link            |
          ==========================
          | function value  (if a  |
          | function)  2 words     |
          ==========================
          | local variables        |
          |////////////////////////|
          ==========================
          | oversized value        |
          | parameters             |
          ==========================
          | saved ACs 2 - 15       |
          ==========================
          | size of conformant     |
          | value parameter space  |
          ==========================
          | conformant array value |
AC17 ->   | parameters             |
        ==============================
          | next available stack   |
          | space                  |
```

Figure G-2: Stack Frame

G.2 MECHANICS OF A PROCEDURE CALL

When PASCAL generates a call to a user-defined procedure, the following general sequence is followed:

1. Save the current stack pointer in AC 0.

2. Push the parameters onto the stack (see below, "Parameter Passing").

3. Push the saved stack pointer onto the stack.

4. Copy the current frame pointer into AC 0. This is the dynamic link.

5. Determine the frame pointer of the block lexically enclosing the called procedure, and copy it into AC

   o This is the static link.

6. Call the procedure by means of a PUSHJ.

After the called procedure returns, the stack is restored to its previous state. PASCAL expects the called procedure to preserve all accumulators except ACs 0 and 1. (Note particularly that AC 16, the frame pointer, must be restored.) In the case of a function, the function value is returned in AC 0 (ACs 0 and 1 if the function is of type DOUBLE).

G.3 PARAMETER PASSING

How a parameter is passed depends on what kind of parameter it is and, for a value parameter, how large it is. The five different methods used to pass parameters are:

1. Value parameter by value

2. Value parameter by address

3. Reference (VAR) parameter

4. Procedure/function parameter

5. Conformant array parameter

Each of these methods is described below.

G.3.1 Value Parameter Passed By Value

A value parameter passed by value is simply pushed onto the stack. This method is used for sets, DOUBLE, and any type that fits into a single word, including integer, real, char, pointers, scalars, and small packed arrays and records.

G.3.2 Value Parameter Passed By Address

Any value parameter larger than one word, other than a DOUBLE or set parameter, is passed by address. The address of the parameter, without indirection or indexing, is pushed onto the stack. It is the responsibility of the called procedure to copy the parameter into its own local storage.

G.3.3 Reference (VAR) Parameter

All VAR parameters are passed by address. The address of the parameter, without indirection or indexing, is pushed onto the stack. Thus, it is perfectly safe (and recommended) to access the contents of a VAR parameter using indirection.

G.3.4 Procedure Or Function Parameter

Procedures and functions, as parameters, are passed as two-word descriptors. The first word of the descriptor is the address of the procedure's entry point; the second is its static link. The static link is needed when the procedure is actually called (see above under "Mechanics of a Procedure Call").

G.3.5 Conformant Array Parameter

A conformant array can be of any size; however, in order for parameters to have fixed offsets from the frame pointer, a fixed-size descriptor must be passed. This descriptor contains a fixed part of two words, and a variable part of three words for each conformant dimension.

The fixed part contains the base address of the array in the first word. The second word contains the overall size of the passed array in words.

Each three words of the variable part contain information on one conformant dimension. The second and third words contain the lower and upper bounds, respectively, on the index of the dimension. The first word contains the size of the dimension, which is the upper bound minus the lower bound plus one.

The conformant array descriptor is the same for both value and VAR parameters. It is the responsibility of the called procedure to copy value conformant parameters into its own local storage.

G.4 PARAMETER ACCESSING EXAMPLE

Figure G-3 gives an example of a typical external procedure declaration, followed by MACRO code showing how the various parameters can be accessed.

```
PROCEDURE Test(first: DOUBLE; VAR second: INTEGER; third: INTEGER); EXTERNAL;

        ENTRY   TEST            ;Define as global symbol
TEST:   ...                     ;Save ACs in local storage
        MOVE    1,-2(17)        ;Get value of THIRD
        MOVEI   2,@-3(17)       ;Get address of SECOND
        MOVE    2,-3(17)        ;Get address of SECOND (alternate method)
        MOVE    3,@-3(17)       ;Get value of SECOND
        DMOVE   4,-5(17)        ;Get (2-word) value of FIRST
        ...                     ;Perform computations
        MOVEM   3,@-3(17)       ;Return new value of SECOND
        ...                     ;Restore ACs
        POPJ    17,             ;Return to caller
```

Figure G-3: External Procedure Declaration

Saving the ACs is not shown; they are assumed to be preserved in some static locations, not on the stack. The stack pointer is used for accessing the parameters. (If the ACs are saved on the stack, or if the stack is used for other purposes, then another AC should be set up for accessing parameters.)

Offset 0 (from AC 17) is the return address; offset -1 is the saved stack pointer. Therefore, offset -2 is the last parameter location. Since THIRD is an integer, it takes up only one word, and is accessed with an offset of -2. The next previous parameter, SECOND, is a VAR parameter, passed by address. The address is in the next previous word, at offset -3. FIRST is a double-precision real, taking up two words. Therefore, offset -4 is the second word of the parameter, and offset -5 is the first word.

G.5 CONFORMANT ARRAY EXAMPLE

Figure G-4 shows an example of an external procedure declaration with a conformant array parameter, and a picture of the stack after the call. The base address of the array is at offset -6; the length in words is at offset -5. The lower and upper bounds on the index are at offsets -3 and -2, respectively. The number of elements is at offset -4.

```
PROCEDURE Confar(VAR a: ARRAY[la..ha:INTEGER] OF INTEGER); EXTERNAL;


          ==========================      -----
-6(17)    | array base address     |      fixed
          ==========================
-5(17)    | size of array (words)  |      part
          ==========================      -----
-4(17)    | # elements  (ha-la+1)  |      variable
          ==========================
-3(17)    | lower bound  (la)      |      (per dimension)
          ==========================
-2(17)    | upper bound  (ha)      |      part
          ==========================      -----
-1(17)    | saved stack pointer    |
          ==========================
 0(17)    | return address         |
          ==========================
```

Figure G-4: Conformant Array Parameter

INDEX

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______

Name ______ Date ______

Organization ______ Telephone ______

Street ______

City ______ State ______ Zip Code or Country ______

- Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MRO1-2/L12
MARLBOROUGH, MA 01752

Do Not Tear - Fold Here and Tape

Cut Along Dotted Line