TOPS–20
PASCAL Primer

AA–L314A–TM

September 1983

This document introduces the TOPS–20 PASCAL language. It is intended to be used by programmers who are new to 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

Digital Equipment Corporation
PO Box CS2008
Nashua, New Hampshire 03061
Telephone:(603)884–6660

Central Region

Digital Equipment Corporation
Accessories and Supplies Center
1050 East Remington Road
Schaumburg, Illinois 60195
Telephone:(312)640–5612

Western Region

Digital Equipment Corporation
Accessories and Supplies Center
632 Caribbean Drive
Sunnyvale, California 94086
Telephone:(408)734–4915

digital equipment corporation • marlboro, massachusetts

First Printing, September 1983

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

digital™

DEC	MASSBUS	UNIBUS
DECmate	PDP	VAX
DECsystem–10	P/OS	VMS
DECSYSTEM–20	Professional	VT
DECUS	Rainbow	Work Processor
DECwriter	RSTS	
DIBOL	RSX	

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

CONTENTS

CONTENTS (CONT.)

CONTENTS (CONT.)

FIGURES

TABLES

PREFACE

PRIMER OBJECTIVES

This primer introduces the PASCAL language for the TOPS-20 operating system. It is designed to provide sufficient information about the language so that you can begin writing PASCAL programs.

PASCAL-20 is an extended implementation of the standard proposed for the PASCAL language by the International Standardization Organization (ISO). This manual describes a subset of PASCAL-20, omitting some advanced features of the language. Once you have mastered the concepts in this primer, you should consult the TOPS-20 PASCAL Language Manual for full reference information on the PASCAL-20 language.

INTENDED AUDIENCE

This manual does not attempt to teach programming concepts. It is assumed that you have experience programming in a high-level language or that you are taking an introductory programming course. However, prior knowledge of the PASCAL language is not necessary.

You need not have a detailed understanding of the TOPS-20 operating system, but some familiarity with TOPS-20 is helpful. Also, you should know how to use a text editor to create a file. If you are new to TOPS-20, see the manual Getting Started with TOPS-20 for introductory material.

HOW TO USE THIS DOCUMENT

This manual contains seven chapters. Each chapter (except the first) introduces new concepts in PASCAL that build on material presented in previous chapters. It is recommended that you read the chapters sequentially to take advantage of this structure.

Chapter 1 explains how to develop PASCAL programs on TOPS-20. You will find sample programs throughout this manual. You can enter, compile, link, and run these sample programs on TOPS-20 using the tools for program development introduced in Chapter 1.

Chapter 2 describes the data types used in PASCAL, as well as the use of variables and expressions.

Chapter 3 describes how to define data types and identifiers in your programs.

Chapter 4 describes how to read and write data.

Chapter 5 describes two structured data types: the array and the record.

Chapter 6 describes PASCAL statements, including the assignment statement, the compound statement, and repetitive statements. In particular, it describes the FOR statement, the REPEAT statement, the WHILE statement, the IF statement, the IF-THEN statement, the IF-THEN-ELSE statement, and the CASE statement.

Chapter 7 describes the use of procedures and functions.

A glossary is presented at the end of this primer. The glossary defines many of the terms presented in the text.

FOR MORE INFORMATION

For reference information on the PASCAL-20 language, consult the TOPS-20 PASCAL Language Manual. It provides additional information on using PASCAL under the TOPS-20 operating system.

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 format descriptions enclose items that are optional, for example: WRITE ([[OUTPUT,]] print list) Do not type the double square brackets. WRITE (OUTPUT, 'Enter an integer')
[ ]	Square brackets mean that the syntax requires the square bracket characters. This notation is used with arrays, for example: ARRAY[index]
{ }	Braces enclose lists from which you must choose one item, for example: {TO {DOWNTO}
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, for example, <CTRL/Z>.
Simple_Procedure	In programming examples, all names created by the programmer are printed in lowercase letters with initial uppercase letters.
UPPERCASE LETTERS	Uppercase letters in a command string indicate fixed (literal) information that you must enter as shown.
lowercase letters	Lowercase letters in a command string indicate variable information you supply.
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 is a computer language invented by Niklaus Wirth and named for Blaise Pascal, a French mathematician and philosopher. It is one of several programming languages used to translate symbolic language programs into machine language. Because the PASCAL language contains features suitable for many different programming applications, it has gained popularity as a general purpose language. PASCAL is used widely in educational institutions because its design lends itself toward teaching structured programming techniques.

PASCAL's English-like statements ease the process of creating a logical flow of control. Thus, you can create code that is descriptive of what the program actually does. PASCAL and its data-structuring facilities encourage modular programming. In modular programming, you divide a problem into individual parts. These parts can be handled easily as subprograms within the main program.

Because you must declare all the data you are going to use at the beginning of the program, you must do some initial planning and organization. All the data structures in your program become readily apparent, making programs easy to read and modify.

Some of the commonly used language features of PASCAL include:

- FOR, REPEAT, WHILE statements
- CASE, IF-THEN, IF-THEN-ELSE statements
- INTEGER, REAL, CHAR, BOOLEAN, enumerated, and subrange scalar data types
- ARRAY, RECORD, SET, and FILE structured data types
- READ, READLN, WRITE, WRITELN input and output procedures
- Functions sine, cosine, square root, round, predecessor, and successor

PASCAL-20 is an outgrowth of the standard proposed for the PASCAL programming language by the International Standardization Organization (ISO). It contains all the elements of the proposed standard for PASCAL plus extensions to the language that allow you more flexibility. These extensions, which are described here in this primer and in the TOPS-20 PASCAL Language Manual, include:

- Exponentiation operator
- Double-precision real data type

- Variable names of up to 31 characters including the dollar sign ($) and the underscore (_)
- OTHERWISE clause in the CASE statement
- Character string and enumerated type parameters for the READ and READLN procedures
- Enumerated type parameters for the WRITE and WRITELN procedures

Throughout this document, PASCAL-20 is referred to as PASCAL unless otherwise specified.

This chapter introduces the PASCAL language. Section 1.1 describes the structure of a PASCAL program. Section 1.2 explains how to develop a PASCAL program on the TOPS-20 operating system. Section 1.3 presents a PASCAL program example.

For more information on all aspects of the PASCAL language as implemented for TOPS-20, see the TOPS-20 PASCAL Language Manual.

1.1 THE STRUCTURE OF A PASCAL PROGRAM

The structure of a PASCAL program consists of the following parts:

1. A program heading
2. A declaration section
3. An executable section
4. A period (.)

The program heading contains the name of the program and any external files the program may use for input and output. The declaration section is the place for all the definitions and data declarations that will be used in the program. The executable section contains all the statements that PASCAL will compile for execution. The period (.) marks the end of the program.

The declaration section and the executable section together are considered a block. A block can be the main body of a program, a function, or a procedure. Functions and procedures are described in Chapter 7.

PASCAL allows free formatting of program text. With free formatting, you can place statements anywhere on a line, divide one statement across more than one line, or place multiple statements on one line. However, you cannot split a name or number between lines or with a space. To make a program easier to read, you can indent parts of the program according to the program flow.

You can also place comments anywhere in a PASCAL program. You enclose comments either within braces { } or within a left-parenthesis/asterisk asterisk/right-parenthesis combination (* *). PASCAL ignores the text between the comment indicators. The following are examples of comments:

```
{This is a comment.}
(* So is this. *)
```

PASCAL provides the following delimiters for you to use when creating a program:

- The reserved word BEGIN
- The reserved word END
- A semicolon (;)
- A period (.)

The reserved words BEGIN and END are used to separate functional parts of a PASCAL program. A reserved word is a word already defined in the PASCAL language; you cannot redefine it to denote anything other than its special meaning. BEGIN and END specify the beginning and end of an executable section. They also delimit a compound statement (See Section 6.2).

Generally, each BEGIN must be associated with an END. Therefore, it is good practice to count the number of BEGINs in your program and make sure you have at least the same number of ENDs. The END delimiter is also associated with the RECORD declaration (Section 5.2) and the CASE statement (Section 6.4.3). No BEGINs are necessary in these two instances where ENDs occur.

The semicolon (;) and the period (.) are also delimiters in the PASCAL language. The semicolon separates successive PASCAL statements. It also terminates the program heading and items in the declaration section. You need not place a semicolon directly after the word BEGIN or before the word END, because BEGIN and END are not statements; they are delimiters. The period marks the end of the PASCAL program.

The following sections explain the various parts of a PASCAL program while building a program called Grocery_Bill. The program Grocery_Bill is an interactive program prompting you for the data it needs, performing calculations, and printing the results. Specifically, it performs the following steps:

- Print instructions for entering prices of grocery items
- Read each price and sum the prices to obtain a subtotal
- Prompt for a yes or no answer to the question 'Do you have any coupons?'
- Read and sum the total value of the coupons that are entered
- Subtract the value of the coupons from the subtotal to obtain a total
- Print a total

The entire program Grocery_Bill is listed in Section 1.3.

1.1.1 The Program Heading

PASCAL programs always begin with a heading. The heading consists of:

- The reserved word PROGRAM
- The program's name
- The program's input and output files (if any), enclosed in parentheses
- The semicolon delimiter

The following is the heading for the program Grocery_Bill:

```
PROGRAM Grocery_Bill (INPUT, OUTPUT);  (* Program header *)
```

This heading tells you that the name of the program is Grocery_Bill, and the files the program uses are INPUT and OUTPUT. INPUT and OUTPUT are names known to PASCAL. They specify predeclared (that is, declared by PASCAL) text files. When you run a program that requires user response, these names indicate that the program uses your terminal for input and output.

1.1.2 The Declaration Section

PASCAL requires that you declare all data items in the program. To declare a data item, you give it a name, otherwise known as an identifier, and indicate what it represents. All declarations in a program must appear in the declaration section.

Any constants, variables, functions, or procedures used in the program must be declared in the declaration section before being used in the executable section. Declarations must appear in the following order (if your program uses them):

1. LABEL -- declares labels for use by the GOTO statement
2. CONST -- defines symbolic constants
3. TYPE -- creates user-defined types
4. VAR -- declares variables and their types
5. VALUE -- initializes variables
6. PROCEDURE and FUNCTION -- declare routines

CONST, TYPE, VAR, PROCEDURE, and FUNCTION are discussed in this manual. LABEL and VALUE are described in the TOPS-20 PASCAL Language Manual.

The program Grocery_Bill contains two kinds of declarations and definitions: TYPE and VAR. The first of these is the TYPE definition:

```
TYPE Yes_No = (Yes, No);
```

This TYPE section defines a data type called Yes_No and the two constants, Yes and No, that constitute the values of the type.

The second declaration is the VAR declaration:

```
VAR  Item_Price, Total,
     Coupon_Amount: REAL;
     Subtotal, Coupons: REAL:=0.0;
     Ans: Yes_No;
```

This variable section declares five real variables: Item_Price, Total, Coupon_Amount, Subtotal, and Coupons. In addition, a sixth variable, Ans, is declared to be of the user-defined type Yes_No. The variable Ans can assume either of two values: Yes or No.

1.1.3 The Executable Section

The executable section contains the statements that, when executed, perform the actions of the program. The executable section follows the declaration section and is delimited by BEGIN and END (followed by a period). The following is the executable section of the program Grocery_Bill:

```
                    (* Executable Section *)

BEGIN
(* Print instructions for entering data. *)
WRITELN('Enter cost of each grocery item. One item per line.');
WRITELN('Enter the value 0.0 to terminate list of items.');
   (* Read prices and add each to subtotal until 0.0 is read. *)
   REPEAT
      READLN(Item_Price);
      Subtotal := Subtotal + Item_Price
      UNTIL (Item_Price = 0.0);
WRITELN('Subtotal equals -- $', Subtotal:7:2);
WRITE('Do you have any coupons? Type yes or no and press <RET>. ');
READLN(Ans);
If (Ans = Yes)
THEN
          BEGIN
          WRITELN('Type value of each coupon. One per line.');
          WRITELN('Type <CTRL/Z) after entering all coupons.');
          (* Read and sum amount of each coupon until end of input. *)
          REPEAT
             READLN(Coupon_Amount);
             Coupons := Coupons + Coupons_Amount
          UNTIL EOF(INPUT)
          END;
(* Subtract coupons from Subtotal to obtain Total, and print Total. *)
Total := Subtotal - Coupons;
WRITELN('Pay this amount -- $', Total:7:2)
END.
                    (* End of Program *)
```

Between the words BEGIN and END are statements and procedures that read and write data, change the value of variables, and control execution. Specifically, they are:

- The WRITE procedure, Section 4.3.1
- The WRITELN procedure, Section 4.3.2
- The READLN procedure, Section 4.2.2

- The EOF function, Section 4.4.2
- The REPEAT statement, Section 6.3.2
- The IF-THEN statement, Section 6.4.1

Several input and output procedures are used in the program Grocery_Bill. For example, the first two WRITELN procedures print instructions on your terminal for entering a list of prices. The text within the apostrophes is printed. The third WRITELN procedure prints text followed by the value of a variable. Again, the text within apostrophes is printed. The integers that appear after the variable name Subtotal specify field width. The first integer specifies the total field width, that is, the number of columns occupied by the value being printed. The second integer specifies the number of places to the right of the decimal point in the printed value.

The remaining output procedures in the program print either the text that is specified in apostrophes or text and the value of a variable, in the same manner as the output procedures explained above.

The program Grocery_Bill contains three input procedures. Each reads a value from the terminal and assigns the value to the variable specified in parentheses, for instance:

```
READLN(Ans);
```

This READLN procedure reads a value and assigns it to the variable Ans. Because Ans is of type Yes_No, the value to be read must be either Yes or No. The READLN procedure accepts the answer Yes or No in either uppercase or lowercase characters.

The executable section of Grocery_Bill also illustrates the assignment statement. An assignment statement contains three parts - a variable, the assignment operator (:=), and an expression. Note the following format:

```
variable := expression;
```

The assignment statement causes the variable to assume the value of the expression, for example:

```
Subtotal := Subtotal + Item_Price;
```

This assignment statement adds the current value of Subtotal and Item_Price, then assigns the sum to Subtotal.

Grocery_Bill contains two kinds of control statements: IF-THEN and REPEAT. The IF-THEN statement is a conditional statement. If the expression (Ans = Yes) is true, the statement following the reserved word THEN is executed. The statement following THEN is a compound statement:

```
IF (Ans = Yes)
THEN
  BEGIN
    .
    .
    .

  END;
```

A compound statement specifies that all the statements within BEGIN and END are executed sequentially as a group.

Finally, there are two examples of the REPEAT statement. One example is:

```
REPEAT
    READLN(Item_Price);
    Subtotal := Subtotal + Item_Price
UNTIL (Item_Price = 0.0);
```

The REPEAT statement specifies that the statements between REPEAT and UNTIL be executed in order and terminated when the value of the variable Item_Price equals 0.0. You do not need a semicolon before UNTIL because the UNTIL clause is not another statement; it is part of the REPEAT statement. The second example is:

```
REPEAT
  .
  .
  .

UNTIL EOF (INPUT);
```

This statement, like the previous REPEAT, performs the statements within the REPEAT and UNTIL repeatedly. However, in this example, execution terminates when the function EOF(INPUT) becomes true. EOF (end-of-file) is a predeclared PASCAL function that returns true at the end of an input file. The <CTRL/Z> that you type after entering the values of coupons indicates the end-of-file condition to PASCAL. (The PASCAL file INPUT is associated with your terminal.)

1.2 PROGRAM DEVELOPMENT

This section explains the steps in developing a PASCAL program. Figure 1-1 illustrates the program development process.

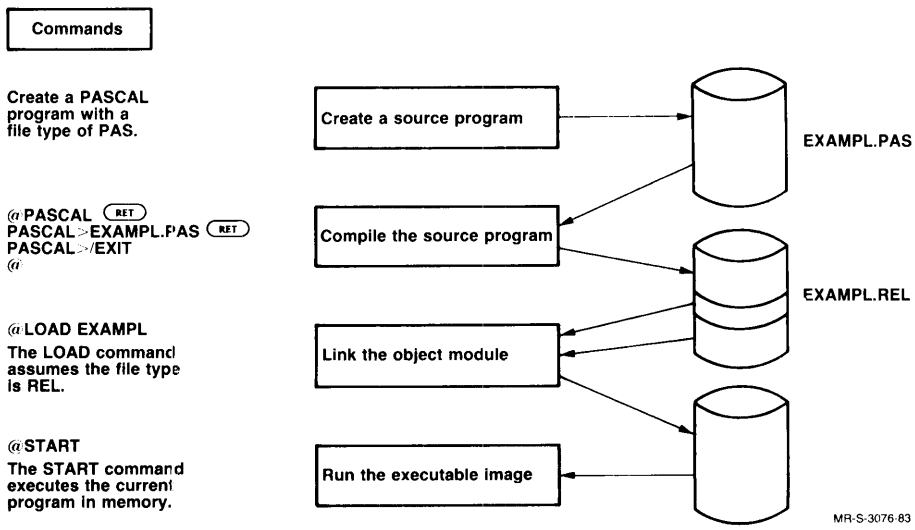

Figure 1-1: Program Development Process

Developing a PASCAL program involves these six steps:

1. Designing the program
2. Creating the source file with an editor
3. Compiling the source program to create an object module
4. Loading the object module to produce an executable image
5. Saving the executable image (optional)
6. Executing the program

After you design the program and create a source file with a text editor, do the following:

1. Call the PASCAL compiler

   @PASCAL(RET)

2. Specify the files to compile

   PASCAL>filename(RET)

3. Exit from the PASCAL compiler

   PASCAL>/EXIT(RET)

4. Load the relocatable binary file

   @LOAD filename(RET)

5. Save the executable image of the file (optional)

   @SAVE filename(RET)

6. Execute the program

   @RUN filename(RET)

Note the following example. The program specified as the file name prints the letter M on the terminal.

```
@Pascal(RET)
PASCAL>letter.pas(RET)
PASCAL>/exit(RET)
@load letter.rel(RET)
LINK:   Loading

EXIT
@save letter.exe(RET)
 LETTER.EXE.1 Saved
@run letter(RET)
 *   *
 ** **
 * * *
 *   *
@
```

All the commands except the PASCAL command include a file name, that is, a TOPS-20 file specification. The PASCAL command returns a prompt:

```
PASCAL>
```

In addition, commands can include optional switches. Command switches modify a command by providing the system with additional information on how to execute the command. For complete information on switches, refer to the TOPS-20 PASCAL Language Manual.

TOPS-20 PASCAL COMPILER

On TOPS-20, you specify the file name of the program you want to compile. You can also call upon the recognition and prompt features to assist you when entering commands. With recognition, you type only enough of the command to identify the command, and then press the (ESC) key. For example, you can type EXE and press the (ESC) key for the EXECUTE command. Pressing the (ESC) key after typing enough to uniquely identify the command causes TOPS-20 to display the rest of the command. In addition, pressing the (ESC) key causes TOPS-20 to display prompts that indicate the next step.

FILE SPECIFICATIONS

The file specification tells the operating system which file to process. A full file specification contains a lengthy string of information. However, because the system assigns appropriate default values to most of the elements in a file specification, you rarely need to specify more than the following two elements:

```
filename.filetype
```

PASCAL recognizes PAS as the default file type. If the file type is PAS, you can specify just the file name. However, if the file has another type (for example, PROGRM.PRG), you must specify the type. The file name identifies the file by its name, and the file type describes what kind of data is in the file. On TOPS-20, the file name and file type of the source file can each contain up to 39 alphanumeric characters. However, the file name of the relocatable binary file must contain no more than 6 characters and its file type no more than 3 characters; otherwise, you cannot load the file with the LINK program. For more information about file specifications and commands, refer to the TOPS-20 User's Guide. For more information about switches that apply to PASCAL, refer to the TOPS-20 PASCAL Language Manual.

1.2.1 Creating a Program

When you write a program, you must create a file, called a source file, that contains the program source statements. You use a text editor to create a source file. For instance, to create a PASCAL program that has the file name EXAMPL and a file type of PAS, you can issue the TOPS-20 EDIT command as follows:

```
@EDIT EXAMPL.PAS (RET)

%File not found, Creating New file
Input: EXAMPL.PAS
00100
```

If the file is a new file, the editor displays an additional message indicating that a new file is being created. You must include a file type (for example PAS) with the EDIT command. The EDIT command invokes 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.

On TOPS-20 in addition to EDIT, you can use any editor to which you have access, for example, the TV editor. For more information about the use of TV, refer to the TOPS-20 TV Editor Manual.

1.2.2 Compiling a Program

After you have edited your file, the next step is to compile it. To compile a source file, issue the PASCAL command. When the PASCAL prompt is displayed, enter the file name, for example:

```
@PASCAL (RET)
PASCAL>EXAMPL (RET)
PASCAL>/EXIT (RET)
@
```

To return to the operating system, type the /EXIT switch to the PASCAL prompt. If the file type is PAS, you can omit the file type because the PASCAL command assumes PAS as the default. However, if you use a file type other than PAS, you must specify the file type.

When you enter the PASCAL command from the terminal, the PASCAL compiler does the following by default:

- Produces an object module that has the same file name as the source file and a file type of REL, for example EXAMPL.REL.
- Uses defaults when it creates the output files (switches on the PASCAL command can override these defaults).

If the compiler does not detect any syntax errors in the source file, the system displays the PASCAL prompt to indicate successful compilation:

```
PASCAL>
```

If the program contains syntax errors, however, the PASCAL compiler displays error messages on your terminal. It is then necessary to edit your source program with a text editor to correct the errors.

TOPS-20 LISTING FILE

On TOPS-20, the /LISTING switch on the PASCAL command requests the compiler to create a program listing. PASCAL does not produce a listing file unless you request one with the /LISTING switch. A program listing includes the program's source statements, line numbers, any error messages or warnings that are reported, and other information related to the compilation. For instance, to create a program listing of EXAMPL on TOPS-20, issue the command:

```
PASCAL>EXAMPL /LISTING (RET)
```

This command causes the compiler to create a file called EXAMPL.LST in addition to the object module EXAMPL.REL. The /LISTING switch does not direct EXAMPL.LST to the line printer.

To obtain a printed copy of the program listing, use the PRINT command as follows:

```
@PRINT EXAMPL.LST
```

A program can be successfully compiled, but still generate warning-level diagnostic messages on your terminal. For example, a number of warning-level diagnostic messages point out the use of nonstandard PASCAL features (that is PASCAL-20 extensions). These messages do not affect the compilation of a program in any way; they are reported only to flag the use of PASCAL-20 extensions. You can suppress the display of diagnostic messages for nonstandard PASCAL features by appending the /NOFLAG-NON-STANDARD switch to the PASCAL command as follows:

```
@PASCAL
PASCAL>EXAMPL /NOFLAG-NON-STANDARD
```

Many of the sample programs in this primer use nonstandard syntax.

1.2.3 Loading an Object File

An object module (for instance, EXAMPL.REL) is not executable. To generate a file that can be executed by the system, you must use the TOPS-20 program LINK. To call LINK, use the LOAD command as follows:

```
@LOAD EXAMPL
```

You can omit the file type because the LOAD command assumes the file type REL by default. The LOAD command loads the object file.

To create an EXE file, use the SAVE command.

```
@SAVE EXAMPL
```

This creates a file named EXAMPL.EXE. EXAMPL.EXE is an executable image, that is, a file containing your program in an executable format.

1.2.4 Executing a Program

To execute the program EXAMPL, use the RUN command or the EXECUTE command. When you issue the RUN command, you provide the name of an executable image; the RUN command assumes a file type (or extension) of EXE by default. Thus, to run the program EXAMPL for which you have created the executable image, issue the RUN command as follows:

```
@RUN EXAMPL
```

The first time you run a program, it may not execute properly. If it has a bug, or a programming error, you may be able to determine the cause of the error by examining the output from the program or the listing file. When you have determined the cause of the error, you can correct your source program and repeat the compiling, loading, and running steps to test the result. You can use PASDDT, the PASCAL debugger, to debug your program. For more information about PASDDT, refer to the TOPS-20 PASCAL Language Manual.

1.3 A PASCAL PROGRAM EXAMPLE

This section presents the complete PASCAL program Grocery_Bill. Note that the characters (* *) mark comments in the program.

```
PROGRAM Grocery_Bill (INPUT, OUTPUT); (* Program header *)

                    (* Declaration section *)

TYPE

    Yes_No = (Yes , No);              (* Defines data type Yes_No
                                         with values Yes and No *)

VAR

    Item_Price , Total,
    Coupon_Amount : Real;             (* Declares three real variables *)
    Ans : Yes_No;                     (* Declares a variable, Ans, of type
                                         Yes_No *)
    Subtotal , Coupons: REAL :=0.0; (* Initializes two real variables *)

                    (* Executable section *)

BEGIN
(* Print instructions for entering data. *)
WRITELN('Enter cost of each grocery item. One item per line.');
WRITELN('Enter the value 0.0 to terminate list of items.');
   (* Read prices and add each to subtotal until 0.0 is read. *)
    REPEAT
       READLN(Item_Price);
       Subtotal := Subtotal + Item_Price
    UNTIL (Item_Price = 0.0);
WRITELN('Subtotal equals -- $' , Subtotal:7:2);
WRITE('Do you have any coupons?  Type yes or no and press <RET>. ');
READLN(Ans);
IF (Ans = Yes)
THEN
        BEGIN
        WRITELN('Type value of each coupon. One per line.');
        WRITELN('Type <CTRL/Z> after entering all coupons.');
        (* Read and sum amount of each coupon until end of input. *)
        REPEAT
           READLN(Coupon_Amount);
           Coupons := Coupons + Coupon_Amount
        UNTIL EOF(INPUT)
        END;
(* Subtract Coupons from Subtotal to obtain Total, and print Total. *)
Total := Subtotal - Coupons;
WRITELN('Pay this amount -- $', Total:7:2)
END.
                        (* End of Program *)
```

You can run the sample program by following the steps outlined in Section 1.2. The source file you create need not have the same name as the PASCAL program. For instance, you can create a file called GROC.PAS to contain the program. The name Grocery_Bill is an identifier known only within the PASCAL program.

If you do not include the /NOFLAG-NON-STANDARD switch on the PASCAL command, the compiler reports warning-level error messages for each nonstandard feature used in the program. For example, the use of underscore (_) characters in Grocery_Bill is a nonstandard feature.

The following is a sample run of the program Grocery_Bill.

```
@pascal
PASCAL>groc.pas/noflag
PASCAL>/exit
@load groc.rel
LINK:      Loading

EXIT
@save groc.exe
 GROC.EXE.2 Saved
@run groc.exe
Enter cost of each grocery item. One item per line.
Enter the value 0.0 to terminate list of items.
1.25
0.57
3.42
1.15
0.89
2.35
0.0
Subtotal equals -- $    9.63
Do you have any coupons?  Type yes or no and press <RET>. yes
Type value of each coupon. One per line
Type <CTRL/Z> after entering all coupons.
0.10
0.25
0.17
0.30
^Z ay this amount -- $    8.81
@
```

The program Grocery_Bill illustrates a small subset of the PASCAL-20 language. It is designed to give you a feel for the way the parts of a PASCAL program work together. The following chapters describe the PASCAL-20 language in more detail.

CHAPTER 2

DATA CONCEPTS

This chapter presents an overview of the data concepts used in PASCAL. Section 2.1 briefly describes the PASCAL concepts of data and types. Section 2.2 describes PASCAL's predefined and user-defined scalar types. Section 2.3 describes structured data types, and Section 2.4 describes variables. Finally, Section 2.5 describes how identifiers and expressions are used in a PASCAL program.

2.1 DATA AND DATA TYPES

Data is a general term for what the computer uses. Within a PASCAL program, you represent data with identifiers; and, in turn, these identifiers represent certain PASCAL constructions. In a PASCAL program, you must assign each identifier a type. The type defines the set or range of values that is associated with the identifier. The following identifiers can be associated with data types:

- Constants
- Variables
- Functions
- Expressions

A constant is a quantity that remains the same throughout the execution of a program. A constant's type is the data type associated with that constant. For example, a value of 3.14 can be associated with the identifier Pi, which is defined to be constant. Because the value 3.14 is a real number, the identifier Pi is evaluated as a real number.

A variable is a quantity that can change during program execution. A variable's type is the data type associated with the set of values the variable can assume. In the program Grocery_Bill in Chapter 1, the identifier Item_Price is an example of a variable; the variable can assume different values in the program. Because Item_Price is defined as an integer, it can be assigned only integer values.

A function is a computation that is associated with a name and that returns a value. A function's type is the same as that of the value it returns. Functions are described in more detail in Chapter 7.

An expression can be a constant, a variable, a function, or a combination of these items separated by operators. Every expression is associated with a type.

PASCAL has three categories of data types:

- Scalar
- Structured
- Pointer

The scalar data types represent ordered groups of values. PASCAL provides two groups of scalar types: standard (or predefined) and user defined. Integers, real numbers, and a subrange of an integer range are examples of scalar data types.

A structured data type consists of a collection of a specific data type. Structured data types are characterized by the types of data that are used, and by their organization. Examples of structured data types are arrays and records.

Pointer data types provide access to dynamic data structures. Dynamic data structures are those in which data is stored nonsequentially. Because pointer types maintain pointers to (that is, addresses of) data, information that is nonsequential can be accessed.

Scalar and structured data types are described in more detail in the following sections. Pointer data types are described in the TOPS-20 PASCAL Language Manual.

2.2 SCALAR DATA TYPES

Scalar data types consist of an identifier that is associated with an ordered set of values. A scalar value is used as an individual item; no part of it can be accessed separately. The number 4 is an example of a scalar value. No part of the number 4 can be accessed; the number 4 is treated as a single entity. PASCAL has two categories of scalar data types: standard and user defined. Standard scalar types are those already defined within PASCAL. User-defined scalar types are data types that you can define within your PASCAL program.

Associated with each data type is a set of operations that can be performed on those values. For example, addition can be done on integers by using the addition operator (+).

The values of a scalar type are ordered; that is, each is either greater than or less than another value of the same type. Thus, you can compare scalar data type values of the same type. For example, among the integers, 2 is greater than 1 but less than 3.

PASCAL provides the ORD function which returns the ordinal value of a character. The ordinal value is the numeric value of the ASCII representation of the character. The format is:

```
ORD(x)
```

The ORD function can be used on any scalar type (including user-defined types) except real numbers. The following example shows the use of the ORD function:

Expression	Returned Value
ORD('A')	65
ORD('a')	97
ORD('3')	51

The term ordinal is used to encompass all scalar values other than those of type REAL; that is, the term ordinal covers integer, character, and Boolean values.

The standard scalar types are:

- INTEGER
- REAL
- CHAR
- BOOLEAN

The type INTEGER is used to represent integers. The type REAL is used to represent real numbers. In addition to the type REAL, PASCAL provides the types SINGLE and DOUBLE representing values that are single- and double-precision real numbers, respectively. The type SINGLE is identical to the type REAL. (Throughout this manual, the term "real type" refers to the REAL, SINGLE, and DOUBLE types collectively.) The type CHAR is used to represent character data. Character data consists of the set of 128 ASCII characters. The ASCII character set is shown in Appendix B. The type BOOLEAN consists of truth values: FALSE and TRUE.

2.2.1 The Type INTEGER

The type INTEGER represents whole number values ranging from (-2**35) to (+2**35)-1 or -34359738368 to +34359738367. You write an integer constant as a sequence of decimal digits; no commas or decimal points are allowed. A minus sign (-) before the number specifies a negative integer. A plus sign (+) can precede a positive integer, but is not required.

Some examples of valid PASCAL integers are:

```
452822
     0
   -17
  +102
-24824
```

PASCAL also accepts integers in binary, octal, or hexadecimal notation. To use binary, octal, or hexadecimal notation, refer to the TOPS-20 PASCAL Language Manual for an explanation of the syntax.

2.2.2 The Type REAL

The type REAL represents decimal numbers. The real numbers are decimal numbers that range from approximately 0.14 * 10**-38 through 3.4 * 10**38, with a typical precision of 8 decimal digits. You can express real constants in two ways in PASCAL:

- Decimal notation
- Floating-point notation

In decimal notation, a constant of the REAL type consists of a minus sign (-) if the number is negative, an integer part, a decimal point, and a fractional part. A plus sign (+) can precede a positive real number, but is not required. At least one digit must appear on each side of the decimal point. Examples of real constants in decimal notation are:

```
 48.25
  0.5
 -0.8
 52.0
  0.0
422.004
```

A zero must precede the decimal point of a fractional quantity, and a zero must follow the decimal point of a whole number quantity.

Floating-point notation is used to represent very large or very small real numbers. In floating-point notation, a constant of the REAL type includes a positive or negative decimal number followed by a positive or negative decimal integer exponent written in exponential notation. For example, the following real constants are written in both floating-point and decimal notation:

Floating-point	Decimal
2.3E2	230.0
0.00023E6	230.0
10.4E-4	0.0010
3.1415927E0	3.1415927
4.5E9	4500000000
-0.4E2	-40.0

The exponent consists of the letter E, which can be read as "times 10 to the power of", followed by a positive or negative whole number. PASCAL prints real numbers in floating-point notation by default.

In floating-point notation, the position of the decimal point "floats" or moves, depending on the value of the exponent. For example, each of the following numbers is equal to 430.0:

```
4.3E2
4300E-1
430E0
```

Note that, if the decimal part of a floating-point number is a whole number, you can omit the decimal point (for example, 430E0).

You can express double-precision real numbers in floating-point notation, replacing the letter E with the letter D. Refer to the TOPS-20 PASCAL Language Manual for details and examples of the type DOUBLE.

2.2.3 The Type BOOLEAN

The type BOOLEAN represents the truth values: FALSE and TRUE. PASCAL orders these values so that FALSE is less than TRUE. Thus, ORD(FALSE) equals 0 and ORD(TRUE) equals 1. Two kinds of operators combine to form Boolean expressions:

- Relational
- Logical

Sections 2.5.2 and 2.5.3 explain how to form Boolean expressions that include relational and logical operators.

2.2.4 The Type CHAR

The type CHAR is used to manipulate character data. A value of type CHAR is a single element of the ASCII character set. The ASCII character set consists of uppercase and lowercase letters, the digits 0 through 9, and various special symbols, such as the ampersand (&). The full ASCII character set is listed in Appendix B.

Appendix B also lists the integer value or ordinal value that corresponds to each element of the ASCII character set. These integers determine how the elements of type CHAR are ordered. For example, the integer 66 corresponds to the uppercase 'B', and 98 corresponds to the lowercase 'b'. Thus, the character 'B' is less than the character 'b'. (All uppercase letters have a lower ordinal value than lowercase letters.)

The ORD function returns the ordinal value for any given ASCII character, for example:

```
ORD('K')
```

This function returns the value 75.

To specify a character value with the ORD function, enclose the value in apostrophes; to specify the apostrophe character, type it twice within apostrophes. Examples of constants of type CHAR are:

```
'A'
'*'
'3'
'b'
' '         (the blank character)
''''        (the apostrophe)
```

The elements of type CHAR are always single characters. A sequence of characters within apostrophes is called a character string (for example, 'John Doe' or 'Memorandum'); character strings are explained in Section 5.1.2.

2.2.5 User-Defined Scalar Types

User-defined scalar types are those types that you define within a PASCAL program. A user-defined scalar type can be either an enumerated type or a subrange of any scalar type except real numbers.

An enumerated type is an ordered set of values that you define by naming an identifier and the values represented by the identifier. The sample program Grocery_Bill in Chapter 1 defines the type Yes_No.

```
TYPE    Yes_No = (Yes, No);
```

The type Yes_No has two values -- Yes and No. Details on user-defined types are presented in Section 3.5.

A subrange type is a specified part of another defined type. The subrange is defined by specifying the lower and upper bounds of the subrange. For example, a subrange of integers could be defined:

```
TYPE    Range = 0..100;
```

2.3 STRUCTURED DATA TYPES

A structured type represents a collection of related data components. Individual components of a structured data type can be accessed and manipulated. PASCAL includes four structured data types: arrays, records, sets, and files.

An array is a group of components of the same type. A record consists of one or more 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.

Chapter 5 presents two structured types: the array and the record. For information about sets and files, refer to the TOPS-20 PASCAL Language Manual.

2.4 VARIABLES

A variable is an entity that can assume a value during program execution. This value can change any number of times during program execution. In PASCAL, every variable has a name, a type, and a value. The value of a variable is undefined until a value is assigned in either the declaration or the executable section of a program.

You establish a variable's name and type in the variable declaration section of a program. The name and type are permanent characteristics during the program execution and therefore cannot be changed. A sample variable declaration section is:

```
VAR Miles, Distance : INTEGER;
    Gallons, MPG, Liters : REAL;
    Measure : CHAR;
    Flag : BOOLEAN;
```

The word VAR signifies the variable declaration part of the declaration section. This variable section declares Miles and Distance as identifiers of the type INTEGER; Gallons, MPG, and Liters as identifiers of the type REAL; Measure as an identifier of the type CHAR; and Flag as an identifier of the type BOOLEAN.

One way to assign a value to a variable is with an assignment statement:

```
Distance := 1043;
Miles := Distance;
```

Because Distance is defined, this statement assigns the value of Distance to the identifier Miles. The value of Miles is then defined.

In addition to assignment statements, you can use variable initializations in the declaration section or input procedures in the executable section to assign values to variables.

A value can be assigned in the VAR section:

```
VAR Miles :  INTEGER := 0;
```

A value can also be assigned in the executable section, as shown in the following example:

```
READLN (Miles);
```

This example uses the READLN statement to assign a value to Miles. In this example, PASCAL is waiting for input from the terminal.

2.5 EXPRESSIONS

An expression is an identifier or group of identifiers and operators that PASCAL can evaluate. The identifiers can represent individual constants, variables, or functions, for example:

```
subtotal
```

The variable name subtotal is an expression that is equal to the current value of subtotal.

Expressions can also be combinations of constants, variables, and functions, separated by operators. For example, the sample program in Chapter 1 includes the following expression:

```
Total := Subtotal - Coupons
```

This expression is equal to the result of subtracting the value of Coupons from the value of Subtotal.

PASCAL uses the following types of operators for forming expressions:

1. Arithmetic operators (such as +, -, /, **)
2. Relational operators (such as <, >, =)
3. Logical operators (AND, OR, NOT)

Every expression has a type. Arithmetic operators are used in arithmetic expressions to produce integer or real values. Boolean results.

2.5.1 Arithmetic Expressions

An arithmetic expression calculates an integer or real value. (For the purposes of this section, the term "real" refers to the REAL type. The rules for using values of type DOUBLE in arithmetic expressions differ from those for type REAL. See the TOPS-20 PASCAL Language Manual for information on using the type DOUBLE in expressions.)

An expression can be an integer or real constant, variable, or function. Alternatively, it can be formed by combining numeric constants, variables, and functions with one or more arithmetic operators (shown in Table 2-1). For example, the following expression consists of two identifier names and the division operator (/):

```
Miles / Gallons
```

This expression equals the value of dividing Miles by Gallons.

Table 2-1: 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
REM	A REM B	Produce the remainder after dividing A by B (can be used when B is <= 0)

The addition, subtraction, multiplication, and exponentiation operators (+, -, *, and **) 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 division operator (/) can be used on both real and integer values, but it always produces a real result.

The DIV operator can be used only with integer values and always produces integer results. DIV divides one integer by another and it drops any remainder.

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. The REM operator can be used whether the integers are equal to, less than, or greater than 0. The REM operator also retains the sign of the dividend.

Table 2-2 shows possible combinations of arithmetic operands and operators and the type of the result.

Table 2-2: Result Types for Arithmetic Expressions

Operator Group	Operand Types (1)	Result Type (1)	Example	Result
+, -, *, **	I op I	I	4 + 5	9
(addition, subtraction, multiplication, exponentiation) (2)	R op I	R	4.2.** 2	1.764E+01
	I op R	R	4 * 4.5	1.800E01
	R op R	R	2.2 - 40.12	-3.792E+01
/	I op I	R	4/2	2.000E+00
(division)	R op I	R	3.2/2	1.600E+00
	I op R	R	4/2.14	1.869E+00
	R op R	R	3.2/2.2	1.455E+00
DIV, MOD, REM	I op I	I	42 DIV 5	8
(division with truncation, remainder)			4 DIV 5	0
			32 MOD 5	2
			-4 REM 3	-1

(1) The symbols "I" and "R" stand for INTEGER and REAL, respectively; the symbol "op" stands for "operator."

(2) When you raise an integer to the power of a negative integer you can get unexpected results. Refer to the TOPS-20 PASCAL Language Manual for the rules of how PASCAL evaluates expressions containing negative integer exponentiation.

2.5.2 Relational Expressions

A relational expression tests whether a specified relationship between two values is valid. It returns TRUE if the relationship holds and FALSE otherwise. For example, to test whether the variable MAX is greater than the value 100, you can use the following expression:

```
MAX > 100
```

A relational expression consists of two scalar or character string variables or expressions (such as MAX and 100 above), separated by one of the relational operators listed in Table 2-3. The operands must be of the same type; you compare characters with characters (single or strings) and numeric values with numeric values (integer or real).

Table 2-3: Relational Operators

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

With character comparisons, the relationship is determined by the ordinal values in the ASCII character set (Appendix B).

Note that in the 2-character operators (<>, >= and <=) the operators must appear in the specified order and cannot be separated by a space.

Relational expressions are often used as tests in PASCAL's conditional and repetitive statements (see Sections 6.3 and 6.4, for example:

```
IF Measure = 'g' THEN
       BEGIN
         .
         .
         .
       END;
```

The statements within BEGIN and END are executed only if the expression (Measure = 'g') evaluates to TRUE.

As another example, suppose you want to compare the values of two integer variables. To determine whether a variable named New_Int is greater than or equal to a variable named Large_Int, you can use the following expression:

```
New_Int >= Large_Int
```

If Large_Int holds the value 64 and New_Int is 72, the expression evaluates to TRUE.

Because the elements of scalar types are ordered, you can form relational expressions using scalar constants as operands. For example, the following expressions are valid:

Expression	Result
'C' < 'R'	TRUE
TRUE > FALSE	TRUE
5 = 4	FALSE

Any expression that contains relational operators or logical operators is called a Boolean expression because it produces a Boolean result.

2.5.3 Logical Expressions

You can form logical expressions by combining Boolean values and the logical operators listed in Table 2-4. Logical expressions return a value of type BOOLEAN.

Table 2-4: 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 Boolean values to form a logical expression. The NOT operator reverses the truth value of an expression; so that if A is true, NOT A is false, and vice versa.

The following examples show logical expressions and their Boolean results.

Expression	Result
(4 > 3) AND (18 = 3 * 6)	TRUE
(3 > 4) OR (18 = 3 * 6)	TRUE
NOT (4 <> 5)	FALSE

Boolean variables and functions can be operands in logical expressions, for example:

```
Flag AND ODD(I)
```

Suppose Flag is a Boolean variable. ODD(I) is a function that returns TRUE if the specified integer is odd and FALSE if the integer is even. Both operands, Flag and ODD(I), must be true for the expression to return a value of TRUE.

Another example using a Boolean expression is:

```
(Ints_Read = 10) OR EOF(INPUT)
```

The EOF(INPUT) function returns TRUE if the end of the file INPUT has been encountered. If either or both of the operands in this expression are true, the expression returns a value of TRUE.

2.5.4 Precedence Rules for Operators

When evaluating expressions containing more than one operator, PASCAL follows rules of precedence to determine the order in which operations are to be performed. An operation with higher precedence is evaluated before an operation with lower precedence. Consider the following expression:

```
A/B + 3*4
```

The division operation is performed first; the multiplication operation is performed second; and the addition operation is performed third. For example, if A equals 4 and B equals 2, A/B is evaluated to return 2.0. Then, 3 is multiplied by 4 to return 12. Finally, the results of these calculations are added together to obtain 14.0.

You can combine operators to form complicated expressions. For example, if all of the operands are integer, the following expression is valid:

```
A + 5 DIV 2 * 4 - C * 3
```

If the current values of A and C are 3 and 8, respectively, this expression evaluates to -13. That is, it is evaluated as if it were written:

```
A + ((5 DIV 2) * 4) - (C * 3)
```

Table 2-5 lists the order of precedence of arithmetic, relational, and logical operators, from highest down to lowest. Those operators on the same line in the table have equal precedence.

Table 2-5: Precedence of Operators

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

In addition, the following rules apply:

1. Expressions enclosed in parentheses are evaluated first regardless of the precedence of operators.

2. Two operators of equal precedence (such as DIV and *) are evaluated from left to right.

The following expressions are evaluated differently, because in the second expression parentheses enclose an addition operation.

Expression	Result
4 + 8 ** 2 DIV 7	13
(4 + 8) ** 2 DIV 7	20

In the first expression, PASCAL performs the exponentiation (**) and integer division (DIV) operations before the addition operation (+). In the second expression, the parentheses force PASCAL to add 4 and 8 first; then the result (which is 12) is squared to obtain 144; and finally the DIV operation is performed to obtain 20.

You should use parentheses when you combine relational and logical operators because the logical operators have higher precedence than the relational operators. For example, in the following expression, the logical operator AND has the highest precedence:

```
A < X AND B <= Y + 1
```

PASCAL attempts to evaluate this expression as if it were written:

```
A < (X AND B) <= Y + 1
```

An error occurs because X and B are not of type BOOLEAN. To insure that the expression is evaluated as you intended, you must enclose the relational expressions in parentheses as follows:

```
(A < X) AND (B <= Y +1)
```

Similarly, you must include parentheses in the following expression:

```
NOT (4 <> 5)
```

Without the parentheses, the expression is evaluated as:

```
(NOT 4) <> 5
```

Because 4 is not a Boolean value, PASCAL generates an error.

Parentheses also help to clarify an expression. A long expression is easier to read if it contains parentheses indicating which operations are to be performed first, for example:

```
A + ((5 DIV 2) * 4) - (C * 3)
```

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

CHAPTER 3

DECLARATIONS AND DEFINITIONS

Every data item used in a PASCAL program must either be declared in the program or already be defined by PASCAL. All declarations and definitions must appear in the declaration section. The declaration section can contain the following parts or sections:

1. LABEL -- declares labels for use by the GOTO statement
2. CONST -- defines symbolic constants
3. TYPE -- creates user-defined types
4. VAR -- declares variables and their types
5. VALUE -- initializes variables
6. PROCEDURE and FUNCTION -- declare routines

These sections are optional in a program. However, when they are included, they must appear in the order listed above. Furthermore, each section (except PROCEDURE and FUNCTION) can appear only once in a declaration section. Thus, each block uses the following reserved words only once: LABEL, CONST, TYPE, and VAR. A block can be a program, a function, or a procedure.

All of these sections introduce symbolic names that represent data items. Section 3.1 describes the use of symbolic names and identifiers; Section 3.2 explains the CONST section; Section 3.3 describes the TYPE section; Section 3.4 describes the VAR section; and Section 3.5 describes how to create user-defined scalar types. The LABEL and VALUE sections, along with the GOTO statement, are described in the TOPS-20 PASCAL Language Manual.

3.1 SYMBOLIC NAMES

Symbolic names are the words used in a PASCAL program: some symbolic names are already defined within the PASCAL language; other symbolic names can be created by the user. For example, the following line of a PASCAL program contains three symbolic names:

```
VAR Ans : Yes_No;
```

The name VAR is defined by PASCAL; the variable name Ans and the type name Yes_No are created by the programmer.

There are three classes of symbolic names in PASCAL:

1. Reserved words
2. Predeclared identifiers
3. User identifiers

Section 3.1.1 describes reserved words and predeclared identifiers. Section 3.1.2 describes how to form user identifiers.

3.1.1 Reserved Words, Semireserved Words, and Predeclared Identifiers

Reserved words, semireserved words, and predeclared identifiers are defined by PASCAL and have a special meaning for the compiler.

PASCAL sets aside certain reserved words that cannot be redefined. Some of the reserved words are:

AND	ELSE	NOT	THEN
ARRAY	END	OR	TYPE
BEGIN	FILE	PROCEDURE	UNTIL
CONST	FUNCTION	PROGRAM	VAR
DIV	IF	RECORD	WHILE
		REPEAT	

Appendix A contains a complete list of the PASCAL reserved words. Reserved words in this text are printed in uppercase letters.

In PASCAL, the following words are considered semireserved words:

MODULE
OTHERWISE
REM
VALUE

Like 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 defined.

PASCAL declares certain identifiers to name types, constants, procedures, and functions. In contrast to reserved words, you can, if necessary, redefine predeclared identifiers for another purpose.

If you choose to redefine these identifiers, you should do so with caution. Once a predeclared identifier is used to denote another item, it can no longer be used for its original purpose within the same program. For example, it is valid to create an identifier named COS within a program. However, once COS is redefined, it would no longer perform the cosine function within the program.

The predeclared identifiers that have been mentioned so far in this text include:

BOOLEAN	INPUT	REAL
CHAR	INTEGER	SINGLE
COS	OUTPUT	TRUE
DOUBLE	READ	WRITE
EOF	READLN	WRITELN
FALSE		

Appendix A presents a complete list of PASCAL predeclared identifiers.

3.1.2 User Identifiers

User identifiers are the names you create to represent programs, constants, variables, procedures, functions, and user-defined types. User identifiers are all the names in a PASCAL program that are not reserved words, semireserved words, or predeclared identifiers.

When forming an identifier, you must follow PASCAL syntax rules. An identifier can be a combination of uppercase and lowercase letters, digits, dollar sign ($) characters, and underscore (_) characters, with the following restrictions:

1. Every identifier must start with a letter, an underscore, or a dollar sign; that is, with any printing character other than a digit.

2. Every identifier must be unique within its first 31 characters.

3. An identifier must not contain any blanks.

4. Uppercase and lowercase letters are considered equivalent.

Although identifiers can be any length, PASCAL only recognizes the first 31 characters. Therefore, two different identifiers that have the same first 31 characters are interpreted as the same identifier.

Because you can use any letter or digit in identifiers, you can easily create names that indicate what the data item represents. This tends to make your programs easier to read and understand. For example, although the word Slug is a valid identifier, it would not be clear if it were used to represent the result of a square root calculation. An identifier like Square_Root, on the other hand, indicates what kind of data that identifier holds.

Some examples of valid identifiers in PASCAL are:

```
Miles
Liters
Math_Scores
FICA_Tax
```

Examples of invalid identifiers are:

```
ARRAY                   (a reserved word)
1more                   (begins with a digit)
Package#                (contains the special character #)
```

The following two identifiers are valid according to the syntax of PASCAL. However, they will be treated as the same identifier because they are not unique within their first 31 characters:

```
New_Incorporated_System_Manager_Functions
New_Incorporated_System_Manager_Resources
```

3.2 CONSTANT DEFINITIONS

You can define identifiers to represent constant values in the CONST part of the declaration section. Defining constants up front makes your program easier to read and to modify. These identifiers and their corresponding values are called symbolic constants. For instance, suppose a program that adds apples to oranges uses the number 100 to indicate the maximum number of fruit that can be summed. Instead of using the number 100 in the program, you can define an identifier that represents 100 as follows:

```
CONST Max_Fruit = 100;
```

The identifier Max_Fruit is more descriptive of the constant's use in the program than the number 100, thus making the program easier to read.

You can define any number of symbolic constants in the CONST section, but the reserved word CONST can appear only once.

The format of the CONST definition is:

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

The constant name can be any valid user identifier. The value can be an integer, a real number, a character, a character string (see Section 5.1.2), a Boolean constant, or another symbolic constant. Successive constant definitions must be separated with semicolons.

The type of a symbolic constant is the type of its corresponding value. For example, Max_Fruit shown above is of type INTEGER because 100 is an integer.

Once you define a symbolic constant, the constant identifier can be used in place of the value later in the program. You can change the value of a symbolic constant simply by changing the declaration in the CONST section. However, the identifier represents a constant value that cannot be changed with subsequent assignment statements or input procedures.

For example, to define a symbolic constant with the value of 25 as the number of students in a class, you can use the following constant definition:

```
CONST Class_Size = 25;
```

You can now use the identifier Class_Size to represent the number 25 anywhere in your program.

The use of symbolic constants generally makes a program easier to read, understand, and modify. If, in the example above, the size of the class is 28 the next term, you can simply modify the CONST definition as follows:

```
CONST Class_Size = 28;
```

Using a constant this way is easier than changing every occurrence of the value in the program.

Another example of a constant definition section is:

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

This CONST section defines six constant identifiers. The identifier Rain is equal to the Boolean value TRUE. The identifier Year represents the integer 2001, and the identifier Pi represents the real number 3.1415927. The identifier Comma represents the character',', and the identifier Country represents the string United States. 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.

3.3 TYPE DEFINITIONS

You can define types in the TYPE section of a PASCAL program. The TYPE section associates an identifier with a specified set of values. If a data type is used more than once within the program, it is useful to define the data type within the TYPE section, rather than in the VAR section. This allows you to create a structure that can be accessed by more than one identifier. For example, if a program uses an array structure three times, you can define the array structure in the TYPE section, and declare three variables of that array type in the VAR section.

The format is:

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

Each type name is a user identifier that denotes a type. The type definition specifies any valid PASCAL type. The type definition can be either an enumerated type or a subrange type.

An enumerated type lists each of the values associated with the type. The following example declares an enumerated type in the TYPE section:

```
TYPE Days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

In this example, the type Days includes the identifiers Sun through Sat. These are ordered from left to right; that is, the value on the left is less than the value to its right. Enumerated types are ordered starting at zero. In this example, Sun < Mon.

A subrange type uses a subset of a type that is already defined, either by PASCAL or within your program. The subrange type is defined by specifying the lower and upper limits of the subrange. The example below shows a subrange type:

```
TYPE Scores = 0..100;
     Work = Mon..Fri;
```

In this example, Scores includes the values of 0 through 100, and Work includes the values Mon through Fri.

3.4 VARIABLE DECLARATIONS

Every variable in a PASCAL program must be declared before it is used. A variable declaration creates a variable and associates it with an identifier and a type. A variable's value is undefined until a value is assigned in the declaration section or in the executable section.

The format of the VAR section is:

```
VAR variable name [,variable name ...] : type [:=value]
   [variable name [ ,variable name ...] :  type;...]
```

The variable name can be any valid user identifier. The type can be any of the predefined scalar types (REAL, SINGLE, DOUBLE, INTEGER, CHAR, or BOOLEAN), an identifier previously defined in the TYPE section, or a type definition as outlined in Section 3.3. You also have the option of initializing a variable in the VAR section. Note the last portion of the format.

You declare variables in the VAR section of a program, for example:

```
VAR     Miles : INTEGER;
        Gallons, MPG, Liters : REAL;
        Measure : CHAR;
        Ans: Yes_No;
```

This VAR section declares one variable (Miles) of the INTEGER type; three variables (Gallons, MPG, and Liters) of the REAL type; one variable (Measure) of the CHAR type; and one variable (Ans) of type Yes_No. Note that you can declare multiple variables in the VAR section, but the reserved word VAR can appear only once.

More examples of variable declarations are:

```
VAR     Error_Flag, Test : BOOLEAN;
        Initial : CHAR;
        Cost, Retail_Pr : REAL;
        Count, Iterations, I, J : INTEGER;
```

This VAR section declares the variables Error_Flag and Test of type BOOLEAN; the variable Initial of type CHAR; the variables Cost and Retail_Pr of type REAL; and finally, the variables Count, Iterations, I, and J of type INTEGER. Section 3.5 describes how to declare variables of user-defined types. Chapter 5 shows how to declare array and record variables.

3.5 USER-DEFINED SCALAR TYPES

PASCAL provides the predefined scalar types -- INTEGER, REAL, SINGLE, DOUBLE, CHAR, and BOOLEAN. In addition, PASCAL allows you to create user-defined scalar types, for example:

```
TYPE Yes_No = (Yes, No);
```

This user-defined scalar type called Yes_No has two values, Yes and No. In this way, it is similar to the predefined type BOOLEAN which has two values, FALSE and TRUE.

There are two classes of user-defined scalar types:

1. Enumerated
2. Subrange

To define an enumerated type, you list the type's values, separated by commas, in parentheses. For example, the type definition for Yes_No is:

```
(Yes, No)
```

To define a subrange type, you specify the bounds of an interval of an existing ordinal type. For example, the following is a subrange of the type INTEGER:

```
0..100
```

This definition specifies a type consisting of the integers from 0 to 100.

You can define a user-defined scalar type in either of two sections of the declaration section:

1. The TYPE section
2. The VAR section

When you define an identifier in the TYPE section, you associate a type name with a set of values. In the example above, the identifier Yes_No is the name of a type in the same way that the identifier CHAR is the name of a type. You must still use the VAR section to declare a variable of the type defined in the TYPE section, for example:

```
VAR Ans : Yes_No;
```

Because Yes_No is a type that was defined in the TYPE section, you can define more than one variable of the type, as follows:

```
VAR Ans, Ans2, Ans3 : Yes_No;
```

When you define a type in the VAR section, you associate one or more variable names with the set of values of the type. Thus, the following defines a variable of a subrange type:

```
VAR Percentage : 0..100;
```

The variable Percentage can take on the values 0 through 100. Percentage is not a type name; it is a variable name.

Some examples are shown below:

Example 1

```
TYPE Days_Of_Year = 1..366;
     Alphabet = 'A'..'Z';
     Digits = '0'..'9';
     January_Temps = -20..+60;

VAR Days_Off : Days_Of_Year;
    Initial : Alphabet;
    Rating : Digits;
    Average_January : January_Temps;
```

The TYPE section defines four subrange types: Days_Of_Year, Alphabet, Digits, and January_Temps. Note that, for an integer subrange type, the limits can include a leading minus sign (-) or plus sign (+). The VAR section declares the variable Days_Off, which can assume the integer values 1 through 366; Initial, which can assume the character values 'A' through 'Z'; Rating, which can assume the character values '0' through '9'; and Average_January, which can assume the integer values -20 through +60.

Example 2

```
TYPE Days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
     Colors = (Red, Yellow, Blue, Orange, Purple, Green);
     Primary_Colors = Red..Blue;

VAR Week : Days;
    Spectrum : Colors;
    Paints : Primary_Colors;
    Work_Days : Mon..Fri;
    Final_Grade : 'A'..'E';
```

The TYPE section defines the types Days, Colors, and Primary_Colors. The type Primary_Colors is a subrange of the enumerated type Colors. The VAR section declares variables of the types Days, Colors, and Primary_Colors. In addition, the variable Work_Days is declared to be of the subrange type Mon..Fri, and the variable Final_Grade is declared to be of the subrange type 'A'..'E'.

CHAPTER 4

READING AND WRITING DATA

To enter data into a program, the program must perform input operations. To display the results of a program's actions, the program must perform output operations. This chapter describes TOPS-20 PASCAL terminal input and output (I/O); that is, how to read and write data interactively from a terminal.

The TOPS-20 PASCAL Language Manual contains additional information on PASCAL I/O. Specifically, it explains the use of input and output procedures on files other than the predeclared files INPUT and OUTPUT.

This chapter covers the following topics:

- The predeclared text files INPUT and OUTPUT
- The predeclared READ and READLN procedures for data input
- The predeclared WRITE and WRITELN procedures for data output
- The predeclared functions EOF and EOLN

4.1 THE PREDECLARED TEXT FILES INPUT AND OUTPUT

Text files are the basis of input and output, and are passed as program parameters to a PASCAL program. Text files represent an input/output device, such as a terminal or a line printer.

You perform I/O operations on file variables. A variable of the structured type FILE is a sequence of data items, called components, that have the same type. The structured type FILE is described in more detail in the TOPS-20 PASCAL Language Manual.

PASCAL predeclares two standard file variables as text files: INPUT and OUTPUT. A text file is a file that has components of type CHAR and that is divided into lines. The I/O procedures and functions explained in this chapter perform operations on either INPUT or OUTPUT, by default.

PASCAL associates the predeclared file variables INPUT and OUTPUT with your terminal, by default. That is, your terminal is treated as the file INPUT (for reading data) and the file OUTPUT (for writing data). The use of INPUT and OUTPUT is necessary to make your program interactive.

Because INPUT and OUTPUT are predeclared by PASCAL, you must not declare them in the declaration section; however, you must specify them in the heading of each program that uses them. For example, for an interactive program that accepts input data and writes output data (such as the example in Chapter 1), you must specify both files, as follows:

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

You can specify the files in either order.

Some programs require no input from the terminal. For instance, a program that prints a table of the ASCII characters needs only the output capability. Its heading might be:

```
PROGRAM Print_ASCII (OUTPUT);
```

This program heading indicates that the name of the program is Print_ASCII and that the program uses the file OUTPUT.

You can access only one component of a file at a time. Associated with every file is a file position that determines which component can be currently accessed. You can imagine a file's position as a window that moves, through which you can see only one component at a time. A file's current position is the position immediately following the file component that was last read or written.

On TOPS-20, you can alter the search path for the INPUT and OUTPUT files without rewriting your programs. You may want PASCAL to obtain input from a file in your disk area (or some other place) rather than from the terminal. Or, you may want PASCAL to write the data to a disk file rather than to the terminal. You have two options: you can rewrite the PROGRAM heading, or you can use the TOPS-20 DEFINE command.

The TOPS-20 DEFINE command defines a logical name. A logical name is a descriptive word used to establish a search route for locating files in other directories or on other structures. When you define a logical name, you tell the system where, and in which order, to search for a file. The two logical names you use are PASIN: for INPUT and PASOUT: for OUTPUT.

To define a logical name, type the following to the TOPS-20 operating system:

```
@DEFINE PASIN:filename(RET)
```

Because PASCAL associates the logical name PASIN: with the INPUT in your program, PASCAL will search for input from the file you specify.

Likewise, you can direct PASCAL to output data to some location other than your terminal by specifying the following:

```
@DEFINE PASOUT:filename(RET)
```

To remove a logical name you have defined, give the DEFINE command but do not type any definition. After the DEFINE command, type only the logical name, and press RETURN, for example:

```
@DEFINE PASIN:(RET)
```

For more information on logical names and the DEFINE command, refer to the TOPS-20 Users Guide.

4.2 READING DATA

To submit data for a program to process, you need procedures that perform input operations. The use of input procedures allows a program to process different sets of data each time it runs. PASCAL provides the READ and READLN procedures for data input.

By default, the READ and READLN procedures get data from the predeclared file variable INPUT. The READ procedure reads values from a file and assigns them to the variables that are specified as read parameters. The READLN procedure performs a READ operation, then moves to the beginning of the next line of the input file.

4.2.1 The READ Procedure

The READ procedure reads data items from the file variable INPUT and assigns the values that are read to specified variables. Thus, you can use a READ procedure to give values to variables, as shown in the following example:

```
READ (Next_Char);
```

This procedure call causes a character to be read from the terminal and assigned to the character variable Next_Char.

The general format of the READ procedure, when using the default file variable INPUT, is:

```
READ ( [INPUT,] variable [,variable...] );
```

Because the file variable INPUT is the default, you can omit its name from a procedure call to READ.

The variable(s) are the parameters of the READ procedure into which values will be read. At least one variable must be specified. The parameters of the READ procedure can be variables of any of the scalar types, including enumerated types. As described in Section 5.1.2, the READ procedure also accepts character string variables as parameters.

The READ procedure reads values from the terminal until it finds a value for each variable that is specified as a parameter. The first value found is assigned to the first variable in the list; the second value is assigned to the second variable, and so on.

Each variable must have the same type as the corresponding value being read, with the exception that an integer value can be read into a real variable. In the example of the READ procedure below, the variables could be of the types REAL, INTEGER, and INTEGER, respectively:

```
READ (Temp, Age, Weight);
```

The following values could be read into the specified variables:

```
98.6 11 75
```

The variable Temp is assigned the value 98.6, Age is assigned the value 11, and Weight is assigned the value 75.

Note that, in the READ procedure shown above, each input value is separated from the next by a space. Numeric data items, typed at the terminal, must be separated by one or more spaces or tabs, or put on new lines. Because the space and tab are values of type CHAR, this

rule does not apply to character data. If a READ procedure specifies a character variable and encounters a space or a tab, the space or tab is read and assigned to the character variable.

As the result of a read operation, the value of the component in the current file position is assigned to a variable; then the file position is advanced one component.

Example 1

Statements	Input
READ (X,Y);	1 2 3 4
READ (A,B);	

These two READ procedures read the values on the input line into the variables X, Y, A, and B. After they are executed, the variable X equals 1; Y equals 2; A equals 3; and B equals 4. The file position is advanced to the position that immediately follows the value 4.

Example 2

```
READ (Month, Date, Year);
```

If each of these variables is of type INTEGER, the following are valid input values:

```
2    14   1984
```

After the READ procedure is executed, Month equals 2; Date equals 14; and Year equals 1984. Note again that you can separate values to be input with any number of spaces. The values also can appear on different lines as follows:

```
2

14          1984
```

In this example, Month is assigned the value of 2; Date is assigned the value of 14; and Year is assigned the value of 1984. The READ statement reads past the EOLN mark and assigns the next input values to the next variables.

Example 3

```
READ (Char_Var);
IF Char_Var <> ' ' THEN
  Count := Count + 1;
```

Assume that Char_Var is a variable of type CHAR and that this segment of code is within a repetitive loop. This program fragment counts the number of characters other than the space character on a line. The READ procedure reads a character and assigns it to Char_Var. If the character is not a space (' '), the variable Count is incremented by one. If the character is a space, the assignment statement (Count := Count + 1;) is skipped.

When reading text files into string variables, spaces are assigned to the string variables when the EOLN mark is encountered. EOLN is then TRUE. A READLN statement must be used in this case to pass the EOLN.

4.2.2 The READLN Procedure

An alternative form of the READ procedure is the READLN procedure. The READLN procedure performs a READ, and then positions the file position pointer at the beginning of the next line, for example:

```
READLN (Miles);
```

This READLN procedure reads a value into the variable Miles, then positions the file position pointer at the beginning of the next line. Thus, any remaining data on the input line is ignored.

In contrast to the READ procedure, the file position pointer is advanced to the first component of the next line at the end of the READLN procedure.

The format of the READLN procedure using the default file INPUT is:

```
READLN ( [[INPUT,]] [[variable]] [[,variable]] ...);
```

After a value is read for each variable that is specified as a parameter, the rest of the current line is discarded; and the file position is set to the first component of the next line.

As shown in the format description, the variable list in the READLN procedure is optional. Therefore, you can use READLN as follows:

```
READLN;
```

This statement advances the file position pointer to the beginning of the line after the current line without reading any values.

Example 1

Statements	Input
READLN (X,Y);	1 2 3 4
READLN (A,B);	7 22 18 12

The values assigned in this example are:

```
X = 1          Y = 2
A = 7          B = 22
```

The first READLN procedure reads the values 1 and 2 and assigns them to X and Y, respectively. Then, the file position pointer is advanced to the beginning of the next line, and the remaining numbers on the first line are ignored. The second READLN procedure starts reading data from the second line of the input file and assigns the value 7 to A and the value 22 to B. If these READLN procedures were both READ procedures, only the first line of input would be read.

Example 2

Statement	Input
READLN (X,Y,Z)	1 100<EOLN> 1000 1001<EOLN>

The values assigned in this example are:

```
X = 1
Y = 100
Z = 1000
```

This procedure call assigns 1 to X, 100 to Y, and 1000 to Z. Then, the file position pointer is advanced to the beginning of the line following the values 1000 and 1001. Note that the READLN procedure reads across lines until a value is found for each specified variable, and moves to the next line only after those values are assigned to the variables. Thus, in this example, when there are no more values on the line containing 1 and 100, the value 1000 from the next line is read and assigned to the variable Z.

Example 3

```
READ (Char_Var);
IF EOLN THEN READLN;
IF Char_Var <> ' ' THEN
   Count := Count + 1;
```

The following shows the value of Char_Var after reading the sample input:

Input	Value of Char Var
this <EOLN>	4
this is <EOLN> a test <EOLN>	11

This example shows a code fragment similar to the one used in Example 3 in Section 4.2.1. Included also in this example is a READLN statement. This program fragment counts the number of characters other than the space character in a file. The READ procedure reads a character and assigns it to Char_Var. If the character read is the EOLN, then the file position pointer is moved past the EOLN mark and positioned at the beginning of the next line. If the character is not a space, the variable Count is incremented by one. If the character is a space, the assignment statement (Count := Count + 1;) is skipped.

4.3 WRITING DATA

To display the results of a program's actions, the program must perform output operations. PASCAL provides the WRITE and WRITELN procedures for data output. By default, the WRITE procedure writes data onto the file OUTPUT, which is associated with your terminal. The WRITELN procedure performs the WRITE procedure, then positions the file position pointer at the beginning of a new line.

4.3.1 The WRITE Procedure

By default, the WRITE procedure writes data into the file variable OUTPUT. It has the general form:

```
WRITE ( [OUTPUT,] print list);
```

Because OUTPUT is the default, you can include or omit the name OUTPUT in the WRITE procedure call. The print list specifies write parameters, that is, the values to be written. It can contain:

- Expressions of any scalar type
- Character strings enclosed in apostrophes

Multiple parameters in the print list must be separated by commas.

To print the value of a symbolic constant or a variable, you need to specify only the variable name. You can print the result of an arithmetic, relational, or logical expression by specifying the expression in the print list. In addition, you can use the WRITE procedure to print a character string to explain the output. Examples of WRITE procedures with variable, Boolean expression, and string parameters are shown below. For each output line, a blank sign ( ) indicates that the corresponding WRITE procedure prints a space.

Statements	Output
`WRITE (IntVar);`	`12`
`WRITE ((2>3) AND Flag);`	`FALSE`
`WRITE ('IntVar equals', IntVar);`	`IntVar equals 12`

Each output line is shown above as PASCAL would print it. PASCAL automatically provides spacing for various kinds of output. Thus, in the first two examples, the output values (that is, the value 12 and the value FALSE) are printed with a default number of leading spaces. You can control the spacing by specifying the field width as explained below.

The third example shows how to print a character string and a value. A character string is a sequence of characters enclosed in apostrophes (in this example, 'IntVar equals') and separated by commas. The value 12 is printed with a default number of leading blanks.

After a WRITE procedure is executed, the file position pointer is positioned immediately after the last value that was written. Thus, if the three WRITE procedures shown above appeared in three successive program statements, all of the output would appear on the same line.

Field Width

The field width is the minimum number of characters that will be written to the terminal. You can specify a minimum field width for each possible write parameter in the print list. However, without the field width specification, PASCAL uses the default values listed in Table 4-1.

Table 4-1: Default Value for Field Width

Type of Variable	Number of Characters Printed
INTEGER	10
REAL	16
DOUBLE	24
BOOLEAN	16
CHAR	1
enumerated	16
string	Length of string

For example, the default field width for a real value is 16 characters. If the value of a real variable called Average is 5.5, the value is printed as follows:

```
Statement                          Output

WRITE (Average);                    5.50000000E+00
```

Note that real values are printed in floating-point format, by default. The value of Average is written in a field of 16 characters (which includes a leading blank).

You can override the default for a particular value by specifying a field width in the print list. The following is the general form of the field-width specification:

```
write parameter : minimum [[: fraction]]
```

Minimum and fraction must be positive integers. Minimum indicates the minimum number of characters, including padding spaces, that are to be written. Fraction, which is used only with real values, indicates the number of characters to the right of the decimal point.

For example, you may prefer to print the real value of Average in the more readable decimal format. You can include field-width parameters in the WRITE procedure call to do this:

```
WRITE ('The average is',Average:4:1);
```

This statement produces the following output:

```
The average is 5.5
```

The integer 4 indicates that at least four characters will be printed. This count includes the decimal point and a minus sign (-) if the value is negative. If the value is positive, as above, this minimum includes a leading blank.

The integer 1 specifies that one digit will appear to the right of the decimal point. Thus, the WRITE procedure above specifies a field at least four characters wide, with one character to the right of the decimal point.

The following rules apply to designating field-width parameters in output procedures:

1. If the fraction parameter is omitted from a real value, the value is printed 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:

   - Truncates strings and nonnumeric scalar values on the right to the specified field width.

   - Prints integers and real numbers in decimal format using the full number of characters needed for the value, thus overriding the field-width specification.

   - Prints real and double values in floating-point format 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.

Example 1

Statement

```
WRITE ('First number -- ',Number:9);
```

Output

```
First number --         1
```

If Number is an integer variable whose value is 1, this statement prints the text ('First number -- ') followed by eight spaces and the numeral 1. That is, 1 is right-justified in the field of nine characters.

Example 2

Statement

```
WRITE ('This is a test string':12);
```

Output

```
This is a te
```

The text in this example is truncated on the right so that it fits into the field of 12 characters.

Example 3

Statement

```
BEGIN
  Number := 5;
  Average := 5.5;
  WRITE (Number:4,' values averaged to ',Average:3:1);
END;
```

Output

```
5 values averaged to  5.5
```

One WRITE procedure can contain several values and strings as in this example. If Number equals 5 and Average is 5.5, the output shown is printed. Three leading blanks are included before the number 5 to fill the print field, which is 4. Note that the value of Average is printed in a field of four characters, including the leading blank, even though the procedure specifies a field of three characters, thus overriding the field-width specification of 3.

Example 4

Statement

```
BEGIN
   Num1 := 71.1;
   Num2 := 29.9;
   Num3 := 101.0;
   WRITE (Num1:5:1,' and',Num2:5:1,' sum to',(Num1+Num2):6:1);
END;
```

Output

```
71.1 and 29.9 sum to 101.0
```

This example shows an arithmetic expression as a write parameter. The values of Num1 and Num2 (71.1 and 29.9, respectively) are each written in a field of five characters. The expression (Num1 + Num2) is evaluated, and the value (101.0) is printed in a field of six characters.

Example 5

Statements

```
WRITE('First column heading');
WRITE('Second column heading':35);
```

Output

```
First column heading                Second column heading
```

Remember that, after a WRITE procedure is executed, the file pointer is positioned after the last character printed. Therefore, two consecutive WRITE procedures print data on the same line. The first procedure call to WRITE prints the text, leaving the file position after "heading". The second procedure call right-justifies its text in a field of 35 characters.

Example 6

If you specify a variable of an enumerated type as a write parameter, PASCAL prints the constant variable that names its value in uppercase letters. For example, suppose the variable Color is defined as:

```
VAR Color : (Blue,Yellow,Black,Slightly_Pale_Peach_Summer_Sunset);
```

Assume that the value of Color is Yellow. Then this WRITE procedure call:

```
WRITE ('My favorite color is ',Color:33);
```

produces the following output:

```
My favorite color is                                YELLOW
```

Note that yellow is printed right-justified and uppercase, preceded by 27 blank spaces.

When the value of Color is Slightly_Pale_Peach_Summer_Sunset, however, the following appears:

```
My favorite color is   SLIGHTLY_PALE_PEACH_SUMMER_SUNS
```

PASCAL only recognizes the first 31 characters in a variable name. Thus, although the field width specified is wide enough for all 33 characters, only the first 31 characters are printed; and they are right-justified in the print field.

4.3.2 The WRITELN Procedure

An alternative form of the WRITE procedure is the WRITELN procedure. The WRITELN procedure performs the WRITE procedure, then positions the file position pointer at the beginning of a new line. It has the general form:

```
WRITELN [ ( [ OUTPUT,] print list) ] ;
```

Write parameters are specified in the print list in the same manner as in the WRITE procedure. Furthermore, the field-width rules described in that section also apply to the WRITELN procedure.

If you have multiple parameters in the print list, the WRITELN procedure prints all of the values on one line, and then starts a new line. Alternatively, you can omit the print list altogether. This is useful when you want to start a new line or write a blank line to the output file.

Example 1

Statements

```
WRITELN('The value of X is ',X);
WRITELN('The value of Y is ',Y);
```

Output

```
The value of X is          10
The value of Y is          15
```

In the output, the write parameters from each WRITELN procedure appear on different lines. After both WRITELN procedures are executed, the file position pointer is positioned at the beginning of a new line following the output. In contrast, if you use WRITE procedures instead of WRITELN procedures, the output from both of the print lists appears on one line, as follows:

```
The value of X is        10The value of Y is        15
```

The file position pointer is positioned immediately after the value 15.

Example 2

Statements

```
WRITELN('Name:', 'Age:':19, 'Soc. Sec. :':26);
WRITELN;
WRITELN('Socrates', 'Old':15, 'Unknown':24);
```

Output

```
Name:                Age:                Soc. Sec. :

Socrates             Old                 Unknown
```

This example illustrates how multiple parameters in the print list of a WRITELN procedure are printed. All the items in the print list are printed on one line. Then, the file position pointer is advanced to the beginning of a new line. This example also shows how to print a blank line by omitting the print list. The second WRITELN procedure call prints no characters, but creates a new line.

4.4 THE PREDECLARED FUNCTIONS EOLN AND EOF

The EOLN and EOF functions are predeclared PASCAL functions that operate on file variables and yield Boolean results. The EOLN function tests the end-of-line condition. The EOF function tests the end-of-file condition.

4.4.1 The EOLN Function

Text files are divided into lines. Each line ends with a line-separator mark indicating the end of a line. You can test for this end-of-line mark with the EOLN function.

The format of the end-of-line function is:

```
EOLN [ (file variable) ] ;
```

The file variable must be a variable of type TEXT. The file variable is the default file INPUT. You can either specify the name INPUT or omit the file variable altogether, because INPUT is the default.

The function EOLN is true when the file position pointer is at the end of a line, or after the last component on a line has been read. Otherwise, EOLN is false.

After a READ procedure reads the last component on a line, the file position pointer is on the EOLN mark.

In contrast, after a READLN procedure reads the last component on a line, the file position pointer is at the beginning of the next line, that is, past the EOLN mark. Thus, after a READLN is performed, the EOLN function is not true unless the next line is empty. For this reason, the input procedure before an EOLN test is usually a READ, not a READLN.

If a READ procedure specifying a character variable as a parameter encounters the EOLN mark, a space (' ') is assigned to the variable.

To specify the end of a line when typing input at the terminal, you press (RET), the RETURN key. After a READ procedure reads the last character that was typed before the RETURN key, EOLN becomes true.

The following loop shows the use of the EOLN function:

```
WHILE NOT EOLN(INPUT) DO
  BEGIN
     READ (Ch);
     Num_Chars := Num_Chars + 1
  END;
```

The variable Ch is of type CHAR, and Num_Chars is of type INTEGER. This loop counts the number of characters on a line. The WHILE statement causes the loop body to execute repetitively as long as NOT EOLN(INPUT) is true. When the last character on the line is read, EOLN becomes true. The WHILE statement tests for NOT EOLN(INPUT), which is now false. Therefore, the loop is not executed again. The WHILE statement is described in Section 6.3.3.

4.4.2 The EOF Function

Every file ends with an end-of-file mark that you can test with the EOF function. The EOF function is true when the file position pointer is on this end-of-file mark. The EOF mark follows the last EOLN mark in a file.

The format of the EOF function is:

```
EOF [ (file variable) ] ;
```

The file variable can be a variable of any file type. As with EOLN, the file variable INPUT is the default.

As soon as the last line in a file is read, EOF becomes true. At all other times, EOF is false.

You usually use the READLN procedure before an EOF test. If you use a READ procedure, the last component in the file is read; and the file position pointer is on the EOLN mark. EOF is false because the file position pointer has not been advanced to the EOF mark.

When a value is being read into a character variable, EOF is false after a READ procedure. In this case, after the last component in a file is read, the file position pointer is at the EOLN mark. EOF is still false. As a result of one more READ operation, a space (' ') is assigned to the character variable; however, the file position pointer remains at the EOLN mark. It is necessary to use a READLN procedure to position the file position pointer at the EOF mark. Until a READLN procedure is used, spaces continue to be assigned to the character variable.

The diagram in Figure 4-1 represents the characters and the EOLN and EOF marks in a text file.

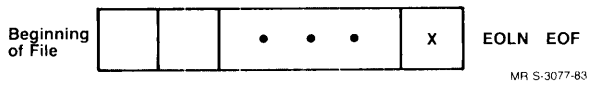

Figure 4-1: The End of a Text File

The symbol X represents the last component of a text file. Suppose the following procedure reads the component denoted by X into a variable:

```
READLN(variable);
```

The variable is assigned the value in X. The READLN procedure advances the file position pointer past the EOLN mark (that is, to the EOF mark). Thus, the file position pointer appears as shown in Figure 4-2.

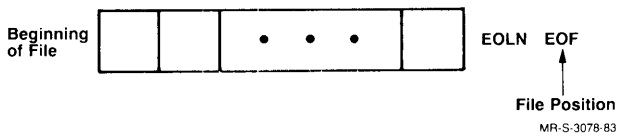

Figure 4-2: File Position Pointer at End of File

When you are typing input at the terminal, you can indicate the end of the file by typing a <CTRL/Z>. The <CTRL/Z> generates an EOLN mark and an EOF mark. When a READLN procedure reads the last component typed before <CTRL/Z>, EOF becomes true. The READLN procedure reads past the EOLN mark and causes the file position pointer to be on the EOF mark.

The following example shows the use of <CTRL/Z> to represent the EOF construct:

```
PROGRAM Average_Score (INPUT,OUTPUT);

VAR
   Score, Total, Count : INTEGER;
   AverageScore : REAL;

BEGIN
   Total := 0;
   Count := 0;
   WRITELN ('Enter your scores.  When done, type CTRL Z');
   WHILE NOT EOF DO
       BEGIN
              READLN (Score);
              Total := Score + Total;
              Count := Count + 1;
       END;
  AverageScore := Total / Count;  (*to produce real result*)
  WRITELN ('The average score is: ',AverageScore:4:1);
END.
```

In Average_Score, the first WRITELN statement displays instructions for entering data items and terminating the list of items with a <CTRL/Z>. Each time the READLN procedure reads a value for the variable Score, it reads past the EOLN mark. On the last iteration, the READLN procedure encounters the end of the file, which was generated by the <CTRL/Z>.

CHAPTER 5

STRUCTURED TYPES: THE ARRAY AND THE RECORD

This chapter describes the use of two structured data types: arrays and records. Structured data types are composed of scalar data types. The types presented in previous chapters of this primer are all scalar types. A variable of a scalar type is an indivisible unit of data. Scalar types cannot be divided into smaller, individually manipulated data items.

A variable of a structured data type, on the other hand, is a collection of related data items that can be accessed and manipulated individually. You can refer to an entire structured variable with one identifier, or you can treat its data items as single variables.

PASCAL provides the following structured types for building data structures:

- Arrays
- Records
- Files
- Sets

An array is a collection of data items of the same type, called components. A record is a collection of data items, called fields, that can be of different types. A file is a sequence of data items, called components, of the same type. A set is a collection of ordinal scalar components or members.

This chapter presents the array (Section 5.1) and the record (Section 5.2) types. A special case of the file type -- that is, the text file -- was introduced in Chapter 4. A detailed presentation of the file and set types, however, is beyond the scope of this primer. The TOPS-20 PASCAL Language Manual describes sets and files in more detail.

5.1 ARRAYS

An array is a group of data items of the same data type. Each data item in the group is called a component of the array. You refer to the whole array with one identifier. You refer to each component with the array identifier and an index that is enclosed in square brackets. The indices need not be integers; they can be values of any scalar type except a real type.

The format of the type definition for an array is:

```
ARRAY [ index type [[ ,index type...]] ] OF component type;
```

The index type can be a subrange of any ordinal type. It can also be the full range of the CHAR type, the BOOLEAN type, or an enumerated type. For example, you can specify the index type in the type definition with the identifier CHAR. However, the index type cannot be the full range of the type INTEGER.

The components of an array can be of any type, including structured types. For example, you can define an array of integers, an array of records, or an array of real numbers. An array of arrays is a multidimensional array, as explained in Section 5.1.1.

The following is an example of an array declaration:

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

An array declaration establishes three properties:

1. The identifier that names the whole array. In the example above, the name of the array is Word.

2. The range and type of the indices. In the array Word, the indices are a subrange -- 1..20 -- of integers.

3. The type of the components. The components of Word are of type CHAR.

You refer to an array component with the array identifier and a bracketed index. The index can be any expression of the index type. Thus, in the array Word, the first component is Word[1]; the second is Word[2]; and so forth. You can use array components in the same way that you use any component of the same type. For example, if Word is defined to be of the CHAR data type, you can use Word[2] as a variable of the CHAR data type. Thus, Word[2] could appear on the left-hand side of an assignment statement or as a parameter of the ORD function.

The type definition shown above can appear in the TYPE section or in the VAR section. Where the type definition appears depends on whether you are defining a type or using a previously defined type. An example of defining an array type in the TYPE section is:

```
TYPE Prices = ARRAY[1..100] OF REAL;
```

This TYPE section defines the type Prices whose index type is the subrange 1..100, and whose component type is REAL. You can declare a variable of type Prices as follows:

```
VAR Items : Prices;
```

Suppose a store has up to 100 kinds of items for sale, and each item is associated with a stock number in the range of 1 to 100. The identifier Items represents the price for each item. Thus, for example, the price for item number 20 is stored in Items[20].

The following declarations show an example of declaring an array variable in the VAR section.

```
TYPE Days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
     WorkDay = 0..24;
VAR WorkWeek : ARRAY[Mon..Fri] OF WorkDay;
```

In the array WorkWeek, the index type Mon..Fri is a subrange of the enumerated type Days. The component type WorkDay is a subrange (0..24) of the integers. This declaration creates the variable WorkWeek in which each of the five components represents the hours worked in one day.

Suppose you want to write a program to calculate the average score of a test of several students enrolled in a course. You can treat the group of test scores as an array, and thus keep each score recorded by a particular student. The following declarations create an array type and a variable of that type:

```
TYPE Tests = 1..Max;
     TestScores = ARRAY[Tests] OF INTEGER;
VAR Score : TestScores;
```

Note that you can use a type identifier (for example, Tests) as the index type in an array definition. If Max is equal to six, TestScores is an array of integers whose index can range from one to six. To use an individual score in an executable statement, specify the array identifier (Score) and an integer expression whose value is between one and Max.

A program that calculates the average of the components in the array Score might be written as follows:

```
PROGRAM New_Aver (INPUT, OUTPUT);

CONST Max = 6;              (* Number of scores to be averaged *)
TYPE Tests = 1..Max;
     TestScores = ARRAY[Tests] OF INTEGER;
VAR Score : TestScores;
    Sum, I, Average : INTEGER;

BEGIN
   Sum := 0;
   FOR I := 1 TO Max DO    (* Access each component of Score *)
   BEGIN
     WRITE('Enter test score: ');
     READLN(Score[I]);     (* Read an integer into each component *)
     Sum := Sum + Score[I](* Sum the components *)
   END;
   Average := Sum DIV Max;
   WRITELN('The followings scores were entered:');
   FOR I := 1 TO Max DO
     WRITELN(Score[I]:4); (* Print each component of Score *)
   WRITELN('The average is: ',Average:3)
END.
```

The program reads each score that is typed after the prompt Enter test score: and calculates a running sum. The average of the scores is the result of the expression Sum DIV Max. Output procedures print each score that was entered and the average score.

A sample run of this program is:

```
Enter test score: 100
Enter test score: 88
Enter test score: 75
Enter test score: 90
Enter test score: 63
Enter test score: 94
The followings scores were entered:
 100
  88
  75
  90
  63
  94
The average is:  85
```

In an executable statement, you can specify a component of Scores with an index that is an identifier. In both FOR loops in the program New_Aver, the current score is denoted by Score[I]. In fact, the index can be any expression of the index type; you could, for example, refer to a component as Score[I+1] (as long as I+1 is in the range 1..6).

The array component Score[I] is used in the same manner as a variable in the READLN statement, in the expression Sum + Score[I], and in one of the WRITELN statements. In fact, Score[I] is a variable of type INTEGER.

You can use the assignment operator (:=) on two arrays of the same type. The following example creates two array variables, called Current_Jan and Record_Jan, that are both of type Month_Temp. Month_Temp is a type that represents the temperatures for each day in a month. The executable section in the example shows the assignment of one array to another.

```
(* Declarations *)

CONST Days = 31;                    (* number of days in Jan *)

TYPE Temp = -20..60;                (* range of temps occurring in Jan *)
  Month_Temp = ARRAY[1..Days] OF Temp;
                                    (* Month_Temp has 31 components,
                                       for each day in Jan *)

VAR Sum, I, Average_Temp,           (* ave temp in current Jan *)
  Record_Ave_Temp : INTEGER;        (* ave temp in Jan with record lows *)

  Current_Jan, Record_Jan : Month_Temp;
                                    (* Current_Jan and Record_Jan
                                       represent each day's
                                       temperature in this year's
                                       Jan and the Jan with lowest
                                       average temp, respectively, *)

(* Executable Section *)

      .
      .
      .

    Sum := 0;
    FOR I := 1 TO Days DO
      Sum := Sum + Current_Jan[I];
    Average_Temp := Sum DIV Days;

    (* if average temp this year is less than the record year,
     assign this year's temp array to the record temp array *)
    IF Average_Temp < Record_Ave_Temp THEN
     Record_Jan := Current_Jan;
```

The last line of the code shows the assignment of one array to another.

This program fragment computes the average of the components of Current_Jan to obtain the average temperature for the month, and assigns that average to Average_Temp. If the value of Average_Temp is less than that of Record_Ave_Temp, the array Current_Jan is assigned to Record_Jan.

5.1.1 Multidimensional Arrays

So far only one-dimensional arrays, that is, arrays with one index, have been discussed. An array whose components are themselves arrays is a multidimensional array. An array can have any number of dimensions. Each dimension has its own index, and each dimension can have a different index type.

For example, the declarations below create a two-dimensional array:

```
CONST ClassSize = 15;
      NumTests = 5;
TYPE Class = 1..ClassSize;
     Tests = 1..NumTests;
VAR ClassScores : ARRAY[Class] OF ARRAY[Tests] OF INTEGER;
```

The variable Class_Scores represents scores on a series of tests for a group of students. If Class_Size is 15 and Num_Tests is 5, these declarations create a two-dimensional array called Class_Scores that can store the scores on 5 tests for each of 15 students.

You can abbreviate the array declaration shown above by specifying all the index types in one pair of brackets as follows:

```
VAR Class_Scores :  ARRAY[Class, Tests] OF INTEGER;
```

To refer to one component of a two-dimensional array, use the array identifier and two indices, one to specify each dimension. The first index corresponds to the first dimension declared, and the second index corresponds to the second dimension declared. The first dimension is the row, and the second dimension is the column. For example, Class_Scores[1,3] indicates the first student's third test score, and Class_Scores[3,1] indicates the third student's first test score.

The Class_Scores array is illustrated in Figure 5-1.

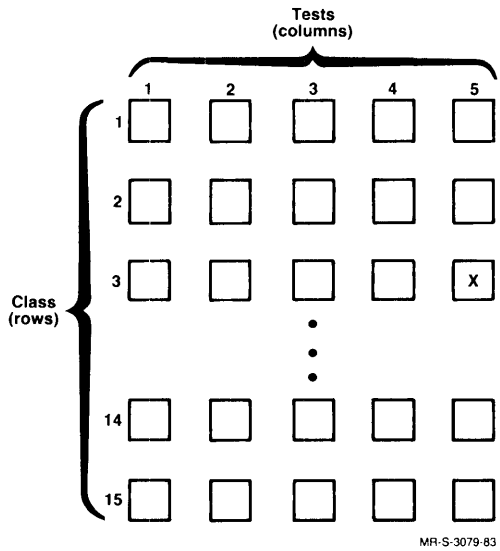

Figure 5-1: The Two-Dimensional Array Class_Scores

The index ranges in Class_Scores -- that is, Class and Tests -- correspond to the rows and columns, respectively, in the figure. In references to one component of Class_Scores, the first index indicates the row, and the second indicates the column. A particular score is found at the intersection of a row of Class and a column of Tests. For example, Class_Scores[3,5], indicated in Figure 5-1 by an X, is at the intersection of the third row and the fifth column.

To access successive components in a multidimensional array, you must use the proper control loop. Nested FOR loops are often used for this purpose. Refer to Section 6.3.1 for a discussion of FOR loops.

The following declaration creates a variable that holds each student's average score:

```
VAR Class_Averages :  ARRAY[Class] OF INTEGER;
```

The following statements include nested FOR loops to compute the average for each student and to store that average in the appropriate component of Class_Averages.

```
FOR I := 1 to ClassSize DO
BEGIN
   Sum := 0;
   FOR J := 1 TO Num_Tests DO
      Sum := Sum + ClassScores[I,J];
   Class_Averages[I] := Sum DIV Num_Tests;
END;
```

The inner FOR loop sums the components in one row (row I). The average of those components is assigned to Class_Averages[I]. The outer FOR loop causes this operation to be performed for each value of I, that is, the values 1 through Class_Size (15).

For example, on the fourth iteration of the outer FOR loop, each component in the fourth row is processed. The components Class_Scores[4,J], where J ranges from 1 to Num_Tests (5), are summed. Class_Averages[4] is assigned Sum DIV Num_Tests, where Num_Tests equals 5.

You can define arrays of three or more dimensions by specifying the appropriate number of index types in the array definition, for example:

```
VAR Hotel_Vacancies :  ARRAY[1..8, 'A'..'B', 1..10] OF BOOLEAN;
```

The variable Hotel_Vacancies represents a hotel with 160 rooms. The hotel has eight stories, each denoted by a number from one to eight. Each story has 2 corridors, and each corridor has 10 rooms. The three dimensions of the array have index types 1..8, 'A'..'B', and 1..10, corresponding to the stories, corridors, and rooms in each corridor of the hotel. Thus, each component in Hotel_Vacancies represents a room in the hotel. An individual component of Hotel_Vacancies has the value TRUE if the room is vacant and FALSE if it is full.

Components Name[1] through Name[12] contain the characters "Joshua Jones". The READ procedure automatically assigns spaces to components Name[13] through Name[20]. Note that, if there are additional characters on the line instead of the EOLN mark, these characters can be read into components of Name.

Similarly, you can use WRITE or WRITELN to print a string variable, for example:

```
WRITE(Name);
```

This WRITE procedure produces the following output:

```
Joshua Jones
```

You can apply the relational operators (<, <=, >, >=, =, and <>) to character strings of the same length. The result of comparing two strings depends on the lexical ordering of the strings. Just as words in a dictionary are arranged according to an alphabetical ordering, character strings are ordered according to the ordinal value of corresponding characters in the string. (See Appendix B for the ordinal value of each component in the ASCII character set.)

PASCAL evaluates string expressions by comparing characters that occupy corresponding positions in the two strings. When the first nonequal characters in the two strings are compared, the string that contains the character with the higher ordinal value is found to be greater than the other string. If all characters are the same, including spaces, the strings are equal.

For example, the following relational expressions are true:

```
'Pekinese' < 'Saint Bernards'
'wine & roses ' > 'wine & cheese'
```

The first expression is true because the ordinal value of 'P' (80) is less than the ordinal value of 'S' (83). When evaluating the second expression, PASCAL compares 'r' and 'c' because these are the first characters that are not the same in the two strings. The ordinal value of 'r' (114) is greater than the ordinal value of 'c' (99).

You can form relational expressions with character string variables as well as with constants. Given the declaration of Section of type Title, that is, PACKED ARRAY[1..20] OF CHAR, the following statement includes a valid relational expression:

```
IF Section = 'Character Strings   ' THEN
   WRITELN('That''s all folks!');
```

5.2 RECORDS

A record is a structured type consisting of related data items of potentially different types. A record is organized into fields; each field can have a different type. An example of a record variable declaration is as follows:

```
VAR Person : RECORD
             Name : PACKED ARRAY[1..20] OF CHAR;
             Age : 0..150;
             Sex : (Female, Male)
             END;
```

The record variable Person has three fields: the field Name is a character string; the field Age is a subrange of integers; and the field Sex is an enumerated type consisting of the values Female and Male. To refer to one field, specify the record variable name and the name of the field, separated by a period (.). Each field can be treated as a variable of the field type, as shown in the following example:

```
Person.Age := 25;
```

Person.Age refers to the field Age contained in the record Person. Person.Age can be assigned a value as if it were an integer variable.

The record type definition format is as follows:

```
RECORD
field name(s) : type;
 [fieldname(s) : type...]
END;
```

As shown, you can specify the names of one or more fields. If there are multiple fields of a particular type, you must separate their names with commas. The type specifies the type of its corresponding field (or fields) and can be any valid PASCAL type. You cannot define more than one field with the same name within a given record.

The reserved words RECORD and END enclose the fields in a record definition. Successive fields of different types must be separated by a semicolon(;). A semicolon is not required between RECORD and the first field name, or between the last field type and END.

NOTE

You do not need a BEGIN with the END of a record definition.

Suppose you are shopping for a new home and you want to maintain information on the houses you see. The factors important in choosing a home might include cost, distance from place of work, number of rooms, method of heating, and location. The following TYPE section defines a record type named House:

```
TYPE House = RECORD
               Cost, Distance : REAL;
               Num_Rooms : 1..20;
               Heat : (Gas, Oil, Electric, Solar, Coal);
               Location : PACKED ARRAY[1..20] OF CHAR;
               Suitable : BOOLEAN
               END;
```

The record type House consists of six fields. Note that you can use a structured type as a field of a record (in the type House, the field Location is a structured type).

To maintain information on a number of houses, you can declare an array of records. For example, if the constant Max_Houses is defined as 10, you can declare an array of 10 House records as follows:

```
VAR House_Choices :  ARRAY[1..Max_Houses] OF House;
```

The variable House_Choices stores multiple records in one array. To refer to one field of one record in this array, specify the variable name House_Choices, an index enclosed in square brackets, a period, and the field variable. This is shown in the following example:

```
House_Choices[I].Heat
```

You use each field of a record variable in the same way a variable of the field type is used. Thus, the following are valid statements:

```
FOR I := 1 TO Max_Houses DO
BEGIN
   READLN(House_Choices[I].Cost);
   READLN(House_Choices[I].Distance);
   READLN(House_Choices[I].NumRooms);
   IF (House_Choices[I].Cost < 70000.0) AND
      (House_Choices[I].Distance < 15.0) AND
      (House_Choices[I].Num_Rooms > 6) THEN
           House_Choices[I].Suitable := TRUE
   ELSE House_Choices[I].Suitable := FALSE
END;
```

You can assign a record variable to another record variable of the same type. For example, the following VAR section declares two record variables of the same type:

```
VAR New_House, Dream_House :  House;
```

If Dream_House is defined (that is, if each field of Dream_House has a value), you can assign Dream_House to New_House as follows:

```
New_House := Dream_House;
```

You can nest records in a record definition; that is, a record can contain a field that is another record, for example:

```
TYPE Employee = RECORD
     Name : PACKED ARRAY[1..20] OF CHAR;
     Address : RECORD
               HouseNo : INTEGER;
               Street, City : PACKED ARRAY [1..20] OF CHAR;
               State : PACKED ARRAY[1..2] OF CHAR;
               Zip : 0..99999
               END; (*end of Address record*)
     EmployeeNo : INTEGER;
     JobTitle : PACKED ARRAY[1..10] OF CHAR;
     Salary : REAL
     END;          (*end of Employee record*)

VAR Employee_N : Employee;
```

To refer to a field within the Address record, you must specify the identifier Employee_N, a period, the identifier Address, a period, and the particular field identifier, as shown in the following example:

```
Employee_N.Address.State := 'PA';
```

This statement assigns the value of the string 'PA' to the State field of Employee_N.Address.

You must read and write the information in records field by field when you are performing I/O operations on text files. PASCAL does not read or write an entire record, for example:

```
WRITELN('Name:',Employee_N.Name,'Number',Employee_N.EmployeeNo);
```

This WRITELN procedure prints two fields of Employee_N: Name and EmployeeNo.

The WITH Statement

When you refer to fields of the same record repeatedly, it is cumbersome to repeat the record name in each reference. The WITH statement allows you to specify the record name once, and refer to the fields directly in the subsequent statement.

The format of the WITH statement is:

```
WITH record variable [,record variable...] DO statement;
```

The record variable specifies the name of the record to which the statement refers. Within the statement, you can refer to a field of the record directly instead of using the record.fieldname format.

For example, the FOR loop that used the record variable House_Choices can be rewritten as follows:

```
FOR I := 1 TO Max_Houses Do
WITH House_Choices[I] DO
BEGIN
  READLN(Cost);
  READLN(Distance);
  READLN(Num_Rooms);
  IF (Cost < 70000.0) AND (Distance < 15.0) AND
     (Num_Rooms > 6) THEN
          Suitable := TRUE
  ELSE Suitable := FALSE
END;
```

Each statement between the BEGIN and END delimiters uses the record name House_Choices[I]. Thus, the following statements are the same:

```
READLN(Cost);

READLN(House_Choices[I].Cost);
```

You can also use the WITH statement to refer directly to fields in nested records. You list the record names in the order in which they are nested, and after the reserved word WITH, for example:

```
TYPE Name = PACKED ARRAY[1..20] OF CHAR;
     Date = RECORD
              Month : (Jan, Feb, March, April, May, June,
                       July, Aug, Sept, 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 := April;
   Day := 13;
   Year := 1743;
   Age := 237
END;
```

The record Hosp contains the field BirthDate, which is also a record (of type Date). The specification of Hosp in the WITH statement allows you to refer to Patient and Age which are fields of Hosp. The specification of BirthDate allows you to access Month, Day, and Year, even though these fields are in a nested record. Thus, the WITH statement shown above is the same as the following:

```
WITH Hosp DO
   WITH BirthDate Do
   BEGIN
      .
      .
      .

   END;
```

The record names in the WITH statement must be specified in the order in which they are nested. For instance, BirthDate is nested within the record Hosp in the declaration; therefore, Hosp must be specified before BirthDate in the WITH statement.

CHAPTER 6

PASCAL STATEMENTS

The basic unit of a PASCAL program is the statement. A statement directs PASCAL to perform an action in a program. A statement consists of a systematic arrangement of reserved words, variables, operators, expressions, and other statements. This chapter describes the following types of statements:

- Assignment statement
- Compound statement
- Repetitive statements
- Conditional statements

The assignment statement gives a value to a variable.

The compound statement, delimited by BEGIN and END, groups other PASCAL statements together for sequential execution as a single statement.

A repetitive statement causes a statement to be executed for a specified number of times or until a certain condition is reached. A conditional statement causes a statement to be executed if a certain value is true.

Repetitive and conditional statements are control statements. Control statements alter the sequence of execution depending on whether specified conditions are met. In the absence of control statements, PASCAL statements are executed in the sequence in which they appear in the source program.

As mentioned in Chapter 1, the semicolon (;) is a delimiter used to separate successive PASCAL statements. As such, it is not needed (although PASCAL does accept it) after a statement followed by a program element that is not a statement. For example, the use of a semicolon before the END delimiter is optional. In this manual, the semicolon is included as part of the statement format descriptions because it is usually necessary and is illegal in only one case. The exception occurs in the IF-THEN-ELSE statement as explained in Section 6.4.2.

6.1 THE ASSIGNMENT STATEMENT

The assignment statement assigns the value of an expression to a variable. The format of the assignment statement is:

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

The assignment statement replaces the current value of the variable with the value of the expression on the right-hand side of the assignment operator. The expression can be any expression having the same type as the variable, with a few exceptions: you can assign an expression of type INTEGER to a variable of type REAL or DOUBLE, and an expression of type REAL to a variable of type DOUBLE. You can also assign to a subrange variable a value of its base type. Note that this statement uses the assignment operator (:=), not the equality operator (=).

For example, if I is declared as an integer variable, the following statement assigns the value 100 to the variable I.

```
I := 100;
```

In addition to constant values, the right-hand side of the assignment statement can be any of the expressions described in Section 2.5.

For example, suppose you have made the following declarations:

```
CONST Yes = 'Y';
      No = 'N';

TYPE Department = (Engineering, Sciences, Math, English,
                   Languages, History, Fine_Arts);

VAR I, Increment : INTEGER;
    Answer : CHAR;
    Grade, Failing_Grade : REAL;
    My_Major : Department;
    Passed : BOOLEAN;
```

Then, the following assignment statements are valid:

```
I := 1;
Failing_Grade := 1.0;
Grade := (4+5+2-1)/3;
Increment := I + 1;
Passed := Grade > Failing_Grade;
Answer := Yes;
My_Major := Fine_Arts;
```

Note that, in the statement Answer := Yes, the expression to be assigned to Answer is a symbolic constant. The value of the symbolic constant Yes is a single character and therefore can be assigned to the CHAR variable Answer. In each of the assignment statements shown above, the type of the expression is the same as that of its corresponding variable.

6.2 THE COMPOUND STATEMENT

You can use the BEGIN and END delimiters to group one or more statements into a compound statement. The statements are executed in sequential order. The format of the compound statement is:

```
BEGIN
    statement1
     [[;statement2...]]     [[ ; ]]
END;
```

The statements within the BEGIN and END delimiters can be any PASCAL statements, including other compound statements. Each successive statement must be followed with a semicolon; however, the use of the semicolon between the last statement and the END delimiter is optional. PASCAL treats the compound statement as if it were a single statement. For example, the following contains a compound statement that is part of an IF-THEN statement:

```
IF Measure = 'g' THEN
    BEGIN
        WRITELN ('Enter the number of gallons.');
        READLN (Gallons);
    END
ELSE
    BEGIN
        WRITELN ('Enter the number of liters.');
        READLN (Liters);
        Gallons := Liters / 3.785; (*converts liters to gallons*)
    END;
```

If the Boolean expression (Measure = 'g') is true, every statement within the first BEGIN and END is executed. If the expression is false, flow of control transfers to the statement following the END and beginning with the ELSE.

This manual uses the term "statement" to mean either a single statement or a compound statement. More examples of compound statements appear throughout this chapter.

6.3 REPETITIVE STATEMENTS

Repetitive statements cause a statement to be executed a specified number of times or until a certain condition is reached. The repetitive statements are:

- FOR
- REPEAT
- WHILE

The FOR statement executes a statement a specified number of times. The REPEAT statement executes a statement and then evaluates a BOOLEAN expression after executing the statement. The statement continues to be executed until the BOOLEAN expression is true. The WHILE statement evaluates a BOOLEAN expression at the beginning of a statement. As long as the expression is true at the beginning of the loop, the statement is executed.

Each of these statements is described in the following sections.

- Because the reserved words REPEAT and UNTIL enclose the statements to be executed, you do not need a compound statement to set off multiple statements. The statements can be delimited with BEGIN and END, but do not have to be. In addition, you need not use a semicolon immediately preceding UNTIL.

The example below shows the use of a REPEAT loop to search for a value in a sorted array. This is an example of a binary search, one of the classic search algorithms. Assume that the program includes the following declarations:

```
CONST Size = 20;      (* dimension of array *)

TYPE Name = PACKED ARRAY[1..20] OF CHAR;

VAR Name_List :  ARRAY[1..Size] OF Name;
    Name_to_Find :  Name;
    I, J, Middle :  INTEGER;
    Found :  BOOLEAN;
```

Assume also that the array Name_List contains an alphabetized list of names. The following program fragment prompts for a name, then searches the array for that name.

```
BEGIN

(* Input the name *)
WRITE ('Name to find : ');
READLN (Name_to_Find);

(* Initialize variables before executing loop *)
I := 1;
J := Size;
Found := FALSE;

 REPEAT
   (* If Name_to_Find is in Name_List, it falls between the
      elements Name_List [I] and Name_List [J] *)
   Middle := (I+J) DIV 2;
   IF (Name_to_Find = Name_List [Middle]) THEN
        BEGIN
           (* Set Found flag and print a message *)
           Found := TRUE;
           WRITELN (Name_to_Find,' is element',Middle:3)
        END
   ELSE IF (Name_to_Find > Name_List [Middle]) THEN
        (* increase lower array bound to select top half *)
        I := Middle + 1
   ELSE IF (Name_to_Find < Name_List [Middle]) THEN
        (* decrease upper array bound to select lower half *)
        J := Middle - 1
UNTIL Found OR (I > J);

IF NOT Found THEN
   WRITELN (Name_to_Find,' is not in the list.');
END.
```

The REPEAT loop contains statements that partition the array and search the appropriate half. The variables I and J initially represent the upper and lower bounds of the entire array Name_List, and are changed during execution to represent the bounds of the part of the array currently being searched. Execution of the loop terminates when Name_to_Find is found (in which case the variable Found is true) or when the value of I exceeds the value of J, indicating that Name_to_Find is not in the array.

For example, if Size is 20, the loop first compares Name_to_Find with Name_List[10]. If their values match, Found becomes TRUE; a message is printed; and the loop terminates.

If Name_to_Find is greater than Name_List[10], that is, if it falls later in the alphabet, then I takes on the value 11. The search is confined to the second half of the array, and the loop is repeated for elements Name_List[11] through Name_List[20].

If Name_to_Find is less than Name_List[10], that is, if it falls earlier in the alphabet, then J takes on the value 9. The search is confined to the first half of the array, and the loop is repeated for elements Name_List[1] through Name_List[9].

On the second iteration, the value of Name_to_Find is compared with the middle element in the selected half of the array, either Name_List[15] or Name_List[5]. If the names do not match, the array is further partitioned. The search continues until Name_to_Find matches an element of Name_List.

If the name is not in the array, eventually the value of I exceeds the value of J, causing execution of the loop to terminate.

Example 2

Assume that Count, Sum, Number, and Average have been declared as integer variables.

```
Sum := 0;
Count := 0;
REPEAT
   READ (Number);
   Sum := Sum + Number;
   Count := Count + 1
UNTIL EOLN(INPUT) OR (Count = 10);
Average := Sum DIV Count;
```

This example reads and sums a list of 1 to 10 integers on a line and averages them. The integers must be entered on one line, and a <RET> must be entered after the last integer. The REPEAT loop reads in one integer, adds it to the sum, and increases Count by 1.

The REPEAT loop terminates when EOLN (INPUT) is true, or when Count equals 10. EOLN (INPUT) becomes true as a result of the <RET> typed after the last integer entered.

Example 3

Suppose that the following declarations have been made in a program's declaration section:

```
TYPE Namestring = PACKED ARRAY[1..20] OF CHAR;

VAR Name : Namestring;
     Namelist : ARRAY[1..30] of Namestring;
     Namecount : INTEGER;
```

The REPEAT loop below uses these variables:

```
Namecount := 0;
REPEAT
    READLN (Name);
    Namecount := Namecount + 1;
    Namelist[Namecount] := Name
UNTIL EOF OR (Namecount = 30);
```

This example reads character strings representing names and stores them in the array Namelist, which contains components of a packed array of characters. Namelist can contain up to 30 names.

The REPEAT loop increases Namecount by 1, then reads one name and assigns it to one element of Namelist (that is, Namelist[Namecount]). The loop terminates when Namecount equals 30 or when EOF becomes true. Because the READLN statement reads one name and then skips to a new line, each name in the input file must be typed on a different line.

6.3.3 The WHILE Statement

The WHILE statement is like the REPEAT statement in that it specifies the repetitive execution of a statement.

The format is:

```
WHILE Boolean expression DO statement;
```

The loop is executed while the Boolean expression is true. When the expression becomes false, execution terminates.

There are three important differences between the WHILE statement and the REPEAT statement.

1. WHILE tests the expression before executing the statement(s) in the loop; REPEAT tests the expression after executing the statement(s). Therefore, if the Boolean expression is false when WHILE is first encountered, the statement following DO is not executed.

2. WHILE controls the execution of only one statement. Hence, to execute a group of statements repeatedly, you must use a compound statement. REPEAT does not require a compound statement; BEGIN and END are optional.

3. WHILE terminates execution of the loop when a condition becomes false. REPEAT terminates execution when the condition becomes true.

Example 1 in the preceding section can be rewritten using a WHILE statement to produce the same results:

Loop with WHILE Statement

```
Sum := 0;
Count := 0;
WHILE NOT EOLN(INPUT) AND (Count < 10) DO
  BEGIN
    READ (Number);
    Sum := Sum + Number;
    Count := Count + 1
  END;
    Average := Sum DIV Count;
```

Loop with REPEAT Statement

```
Sum := 0;
Count := 0;
REPEAT
   READ (Number);
   Sum := Sum + Number;
   Count := Count + 1
UNTIL EOLN(INPUT) OR (Count = 10);
AVERAGE := Sum DIV Count;
```

The differences between the two examples lie in the specification of the conditions for terminating the loop. The WHILE loop is different in these ways:

- The test for EOLN(INPUT) must be written as NOT EOLN(INPUT) so that the loop is repeated as long as EOLN(INPUT) is false.
- The condition determining that only 10 integers can be averaged must be rewritten as Count < 10 (instead of Count = 10). On the last iteration, the tenth integer is read, and Count becomes 10. Count < 10 is then FALSE, so the loop is not executed again.
- The logical expression uses the operator AND instead of OR. The statements are executed as long as both conditions are true.

Example 2

```
WHILE NOT Errorflag AND (Intcount < 100) DO
   BEGIN
     READ (Int);
     IF Int > 0 THEN Poscount := Poscount + 1
     ELSE IF Int < 0 THEN Negcount := Negcount + 1
          ELSE Errorflag := TRUE;
     Intcount := Intcount + 1
   END;
```

This WHILE loop reads an integer; and, if the integer is positive, the variable Poscount is incremented. If the integer is negative, the variable Negcount is incremented. Up to 100 integers can be counted; the number of integers counted is accumulated in Intcount. If a zero is encountered in INPUT, Errorflag becomes TRUE; and the loop is terminated.

Suppose that, in the program surrounding the above program fragment, there is more than one way to obtain an error and thus assign the value TRUE to Errorflag. If Errorflag is true before the program encounters the WHILE statement, the statements within the loop are not executed. Similarily, if Intcount is greater than or equal to 100, the loop is not executed.

6.4 CONDITIONAL STATEMENTS

Conditional statements control the flow of the program depending on the evaluation of an expression. Unlike the repetitive statements, the conditional statements direct only the flow of execution; they do not directly control the number of times that a statement or statements are executed.

The conditional statements in PASCAL are:

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

6.4.1 The IF-THEN Statement

The IF-THEN statement executes a statement or statements only if an expression is true. The format is:

```
IF Boolean expression THEN statement;
```

If the expression is true, then the statement is executed. If the expression is false, program control passes to the statement following the IF-THEN statement.

The reserved word THEN and the statement that directly follows it are called the THEN clause of the IF-THEN statement. The statement following THEN is called the object of the THEN clause, for example:

```
IF A < 0 THEN
   Neg_Ints := Neg_Ints + 1;
```

The object of the THEN clause is

```
Neg_Ints := Neg_Ints + 1;
```

If the value of A is less than zero, the value of Neg_Ints is increased by one. Note that you must not place a semicolon after the reserved word THEN. If you do, an empty statement becomes the object of the THEN clause. In the example above, if there had been a semicolon after THEN, the assignment statement would be executed regardless of the value of the Boolean expression.

Example 1

```
IF (Ans = Yes) THEN
   BEGIN
     .
     .
     .
   END;
```

The object of the THEN clause can be a compound statement. The statements between BEGIN and END are executed if the Boolean expression is true. If it is not, execution continues with the statement following the END.

Example 2

```
IF (Ch >= 'A') AND (Ch <= 'Z') THEN
  BEGIN
    Letter := TRUE;
    LetterTotal := LetterTotal + 1
  END;
```

If both relational expressions are true, the compound statement is executed.

Example 3

```
IF Errorflag THEN
   WRITELN ('Index number out of bounds');
```

The Boolean expression can be a single Boolean variable, as in this example. The WRITELN statement writes the message in apostrophes if the current value of Errorflag is TRUE.

6.4.2 The IF-THEN-ELSE Statement

The IF-THEN-ELSE statement executes one statement if the expression is true, and another statement if the expression is false. The format of the IF-THEN-ELSE statement is:

```
IF Boolean expression THEN statement1 ELSE statement2;
```

Statement1 and statement2 can be any PASCAL statements. Statement1 is executed only if the expression is true. If the expression is false, then statement2 is executed. The reserved word ELSE and statement2 are called the ELSE clause of the IF-THEN-ELSE statement.

NOTE

You must not place a semicolon after statement1 and before the word ELSE. The IF-THEN-ELSE statement is a single statement; the IF-THEN clause cannot be separated from the ELSE clause. Because ELSE does not denote an independent statement, you receive a compile-time error message if there is a semicolon directly before the word ELSE.

The ELSE clause is always associated with the closest IF-THEN statement, for example:

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

The ELSE clause is associated with the second IF-THEN statement. Therefore, if A and B are both equal to 1, 1 is assigned to the variable D. If A is not equal to 1, neither assignment statement is executed. If you want the ELSE clause to be associated with the first IF-THEN, you can write the sequence as follows:

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

The object of the THEN clause of the outer IF-THEN-ELSE statement consists of:

```
BEGIN
  IF B <> 1 THEN C := 1
END
```

And the ELSE clause is:

```
ELSE D:=1;
```

Thus, if A is equal to 1, the THEN clause is executed, and the ELSE clause is ignored. If A is not equal to 1, 1 is assigned to D regardless of the value of B.

Example 1

```
IF (LastInitial >= 'A') AND (LastInitial <= 'M') THEN
   Billdate := 14
ELSE Billdate := 28;
```

This example determines billing dates depending on the initial of a last name. Bills are sent on the 14th of the month to each customer whose last name starts with A through M, and on the 28th to customers whose last name starts with N through Z.

Example 2

```
IF (Card_Sum > 21) THEN
   Lose := TRUE
ELSE IF (Card_Sum >=17) THEN
   Deal := FALSE
ELSE
  BEGIN
     Deal := TRUE;
     Card_Sum := CardSum + Newcard
  END;
```

This example shows a simple strategy for the game Blackjack. Note the nested IF-THEN-ELSE statements that allow the program to select and execute one of a group of statements. In this example, if the cards add up to more than 21, the player loses. If the sum is between 17 and 21, inclusive, the player is not dealt any more cards. If the sum is less than 17, the player is dealt another card. The IF-THEN-ELSE construct can become awkward if there are more than a few selections to be made. A more elegant way to program this type of problem uses PASCAL's conditional CASE statement, which is explained in the following section.

6.4.3 The CASE Statement

The CASE statement selects one of a group of statements for execution. Constant values or case labels are associated in a CASE statement with each possible statement or action to be performed. The format of the CASE statement is:

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

The case selector can be any expression (not only a variable) that evaluates to an ordinal type. The case-label list can consist of one or more values of the same type as the selector, separated by commas. Each label list is associated with the statement to its right. The label list and statement must be separated by a colon (:). You can include an optional OTHERWISE clause that contains a statement that is not associated with a label.

CASE executes the statement labelled by a value that equals a specified expression, for example:

```
CASE Correct_Answers OF
       9,10 : Score := 'A';
          8 : Score := 'B';
          7 : Score := 'C';
          6 : Score := 'D';
0,1,2,3,4,5 : Score := 'F'
END;
```

This CASE statement compares the value of the expression Correct_Answers to the case labels (the numbers 0 through 10). If the value of Correct_Answers is any of the numbers from 0 to 10, the assignment statement to the right of that number is executed.

NOTE

No BEGIN is allowed with the END to a CASE statement.

You can specify the labels in any order. The difference in the ordinal values of the largest and smallest labels must not exceed 1000. Each label can appear only once within a CASE statement.

At run time, if the selector equals one of the specified labels, the statement to the right of that label is executed. If an OTHERWISE clause is included and the selector does not equal one of the labels, the statement next to OTHERWISE is executed.

Setting the CHECK compile option controls whether an error occurs if there is no OTHERWISE clause and none of the case labels are selected. If the option is set, an error occurs at run time. If the option is not set, the statement following the CASE statement is executed. For more information about compile options, refer to the TOPS-20 PASCAL Language Manual.

Example 1

Suppose you have made the following declarations:

```
VAR Month :  (Jan, Feb, Mar, Apr, May, June,
              July, Aug, Sept, Oct, Nov, Dec);
Season :  (Winter, Spring, Summer, Fall);
Temp :  INTEGER;
Snow :  BOOLEAN;
```

You can use the following CASE statement:

```
CASE Month OF
          Jan, Feb, Mar : BEGIN
                          Season := Winter;
                          IF (Temp <= 30) THEN
                              Snow := TRUE
                          END;
          Apr, May, June : Season := Spring;
          July, Aug, Sept : Season := Summer;
          Oct, Nov, Dec : Season := Fall
END;
```

At run time, the current value of Month is evaluated. The statement associated with that value is executed; the rest of the statements are ignored. For example, if Month equals May, then Spring is assigned to the variable Season.

Example 2

This example represents the relationship of combinations of genes to the occurrence of dominant versus recessive traits. Assume that Gene_Combo and Trait are variables of enumerated types declared as follows:

```
VAR Gene_Combo : (Recessive_Recessive, Recessive_Dominant,
                  Dominant_Recessive, Dominant_Dominant);
    Trait : (Recessive, Dominant);
```

These variables are used in the following CASE statement:

```
CASE Gene_Combo OF
     Recessive_Recessive : Trait := Recessive;
     OTHERWISE  Trait := Dominant
END;
```

If the value of Gene_Combo is Recessive_Recessive, the value Recessive is assigned to Trait. If Gene_Combo evaluates to any other value, the OTHERWISE clause is executed: that is, Dominant is assigned to Trait.

CHAPTER 7

PROCEDURES AND FUNCTIONS

The statements described in the previous chapters provide the basic building blocks for writing PASCAL programs. This chapter describes the use of subprograms and the passing of data to and from subprograms.

7.1 SUBPROGRAMS

A subprogram consists of several statements grouped together to perform a single part of a larger task. Subprograms are useful for solving large problems; you can divide the problem into many pieces and solve each piece individually.

For example, if you are writing a program to maintain student records, the program will be easier to write if it is broken into subprograms. The larger task of maintaining records could be broken down into such segments as adding students, deleting students, listing available courses, and entering grades.

Subprograms are also useful when the same code segment is used more than once within the program. Using a call to a subprogram is simpler and clearer than duplicating the same code segment several times within the main body of the program.

To handle these types of problems, PASCAL uses two types of subprograms:

- Procedures
- Functions

Both procedures and functions are declared in the declaration section (see Sections 7.2 and 7.3) and are called from the executable section.

A procedure is a group of statements that perform a set of actions. A function is similar to a procedure in that: it is a set of statements associated with an identifier; and it is declared in the declaration section. However, a function has a type, and it returns a value of that type.

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

You can use predeclared subprograms that are defined by PASCAL and denoted by predeclared variables; or you can create user-declared subprograms, as described in this chapter. Appendix C contains tables of all the predeclared procedures and functions in PASCAL.

7.1.1 Format of a Subprogram

Subprograms are similar in format to main programs. A subprogram consists of a heading and a block; the block contains a declaration section and an executable section. The heading of a subprogram is slightly different from that of a program, because the subprogram heading can contain a formal parameter list. See Section 7.4.1 for a description of formal parameters. For a function, the heading also indicates the type of the value returned. The declaration section defines local data items that are used in the subprogram. The executable section contains the statements that perform the actions of the subprogram.

All subprograms must be declared in the declaration section of the main program or of another subprogram. A subprogram is not executed when it is declared. It is executed as a result of a subprogram call, which can appear in the main program or in another subprogram.

Subprograms are said to be nested within the main program. In addition, subprograms can be nested within other subprograms. A nested subprogram can be called only from inside the block that declares it.

The general format of a subprogram heading includes one of the reserved words FUNCTION or PROCEDURE, followed by the subprogram identifier. An optional parameter list can be included. Functions must also include a result type as part of the heading.

The use of variables is described in Section 7.1.2. Sections 7.2 and 7.3 explain the use of procedures and functions. Section 7.4 provides information about the use of parameters.

7.1.2 Local and Global Variables

Local variables are variables that you define within the subprogram. PASCAL uses local variables only in the subprogram where they are defined. Global variables are those that are declared in the declaration section for the main block of the program. Although the global variable is declared outside a subprogram, it can be accessed by the subprogram. Thus, variables declared in the main program block are global to all subprograms.

A variable declared in a subprogram is global to all its nested subprograms. In addition, predeclared variables are global to all parts of a PASCAL program. Global variables allow you to implicitly pass variables to and from a subprogram to the main body of the program.

Global variables should be used with care in subprograms. When global variables are used, values are not explicitly passed to and from the subprogram. Therefore, debugging and modifying a program is more difficult than if values are explicitly passed. To avoid the problem of accidentally changing values in subprograms, it is suggested that parameters be used to explicitly pass values to and from subprograms. Parameters are described in Section 7.4.

7.1.3 Scope of Identifiers in Subprograms

The scope of an identifier is the part of the program in which you have access to the identifier, that is, the block in which it is declared. Thus, the scope of a variable declared in the main program block is the full program. The scope of a variable declared in a subprogram block is that subprogram and all subprograms nested within it.

In a subprogram, you can redeclare an identifier that has been declared in an outer block. When you use an identifier in a subprogram that is declared both in that subprogram and in an outer block, the identifier always refers to the declaration of most limited scope. The scope of a global variable does not include any block in which it is redeclared. Thus, the local variable can have attributes distinct from those of the global variable (for instance, type or value). You should use great care if you redeclare an identifier in a subprogram. It could easily be confusing if the program needs to be debugged or modified at a later point. Redeclaring an identifier within a subprogram should be avoided if possible.

7.2 DECLARING A PROCEDURE

Procedures are declared in the declaration section of a program. To declare a procedure, you specify its heading and block. You can declare a procedure in the main program or in another subprogram. The format of a procedure declaration is:

```
PROCEDURE procedure name [ [[formal parameters)]] ;
   declaration section
BEGIN
   statement [[;statement...]]
END;
```

The reserved word PROCEDURE is required. The procedure name specifies the identifier to be used as the name of the procedure. The formal parameters describe the parameters used in the procedure (see Section 7.4).

The declaration section can contain local declarations and definitions. The statements between BEGIN and END can be any PASCAL statements. The block in the procedure is identical to the block in the main program, with the exception that the procedure block ends with a semicolon (;) rather than a period (.). You should not redeclare the formal-parameter names or the procedure name as local variables in the procedure.

7.2.1 Calling a Procedure

A procedure is executed as a result of a procedure call. A procedure call consists of the procedure name and, when required, an actual parameter list. If the procedure declaration contains a formal parameter list, the procedure call must contain a corresponding actual parameter list.

The following example shows a program that passes values to and from a procedure.

```
PROGRAM Swap (INPUT,OUTPUT);                    (* this program orders pairs
                                                   of numbers in ascending
                                                   order *)
VAR
        X, Y : INTEGER;

PROCEDURE Switch (VAR A, B : INTEGER);          (* Switch swaps values
                                                   of variables *)
    VAR Temp: INTEGER;

    BEGIN
       Temp := B;
       B := A;
       A := Temp;
    END;

                              (* main program *)

BEGIN
    WRITELN ('This program will order a list of numbers for you.');
    WRITELN ('When you are done, type <CTRL> Z.');
    WRITELN ('Enter two numbers of any size.');
    WHILE NOT EOF DO
        BEGIN
          READLN (X, Y);
          IF X > Y THEN Switch (X,Y); (* here is the procedure call *)
          WRITELN (' ', X, Y);
        END;
    IF EOF THEN WRITELN ('END OF LIST');
END.
```

The procedure Switch is defined in the declaration section at the beginning of the program. The call to the procedure is in the main block of the program:

```
IF X > Y THEN Switch (X,Y);
```

If the expression (X > Y) returns a value of TRUE, the procedure Switch is called and executed. The actual parameter list names the variables (X and Y) that are passed to the subprogram. The formal parameter list in the procedure heading names the variables (A and B) to which the values from the main block are assigned.

7.2.2 Procedure Example

The following example shows the program Simple_Procedure. This program includes a procedure to perform a simple output operation.

Execution begins in the main body of the program with the WRITELN statement. Execution continues to the evaluation of the IF-THEN statement. If the statement is true, that is, if Answer equals yes, then the procedure Read_Fortune is called. The procedure is called by using the procedure name as a statement.

When the procedure Read_Fortune is called, the statements in the procedure are executed. Because no values are being passed to or from the procedure, it is not necessary to specify any parameters when the procedure is called or in the procedure definition. After the statements in the procedure have executed, the flow of the program returns to the statement following the call to the procedure. In this example the final WRITELN statement is executed following the procedure execution.

```
PROGRAM Simple_Procedure (INPUT, OUTPUT);

TYPE     Yes_No = (Yes, No);            (* defines data type with values
                                           yes and no *)

VAR      Answer: Yes_No := Yes;         (* defines one variable of
                                           user-defined type Yes_No and
                                           assigns value Yes *)

                                        (* this procedure executes the
                                           WRITELN statements, when this
                                           procedure is called *)
PROCEDURE Read_Fortune;
BEGIN
    WRITELN ('You will be healthy, wealthy, and wise');
END;

                        (* main body of the program *)

BEGIN
    WRITELN ('Would you like to see your fortune?');
    WRITELN ('Please answer yes or no.');
    READLN (Answer);
    IF Answer = Yes THEN Read_Fortune
     ELSE
       WRITELN ('If you had typed "yes" you would have seen',
                'your fortune.');
    WRITELN ('This is the end of the program.')
END.
```

7.3 DECLARING A FUNCTION

A function is a group of statements that is associated with a name and that returns a value. PASCAL includes in the language such functions as SQR, SQRT, SIN, COS, and ARCTAN. In addition, in PASCAL you can define functions within your programs.

Functions are similar in format to procedures. However, a function is associated with a type, and returns a value of that type. The type of the value returned is specified in the function heading.

To declare a function, specify its heading and block in the procedure and function part of a declaration section. The format of a function declaration is identical to that of a procedure, except that the function heading begins with the reserved word FUNCTION instead of PROCEDURE, and the heading includes a result type. The format is:

```
FUNCTION function name [[(formal parameters)]] : result type;
   declaration section
BEGIN
   statement [[;statement...]]
END;
```

The function name is the identifier used as the name of the function. The formal parameter list must conform to the format described in Section 7.4. The result type must be a scalar or pointer type.

The block of a function is the same as the block of a procedure. Both functions and procedures are terminated with a semicolon (;).

Every function must include one or more statements that assign a value of the result type to the function name. In the function Divides (Section 7.3.2), if the Boolean expression -- (Dividend REM Divisor = 0) -- is true, TRUE is assigned to the function name. Otherwise, FALSE is assigned to the function name. If a value is not assigned to the function name during execution of the function, the function result is undefined.

7.3.1 Calling a Function

A function is executed as a result of a function call. A function call is an expression in which the function and the formal parameters are included. Because the function call is an expression, it can be used anywhere other expressions can be used. For example, a function reference can appear on the right-hand side of an assignment statement.

Unlike a procedure call, a function reference is not a statement in itself. However, actual parameters in a function reference are used in exactly the same manner as in procedure calls; they are passed to the function.

7.3.2 Function Example

The following is a valid function declaration:

```
FUNCTION Divides (Dividend, Divisor : INTEGER) : BOOLEAN;
BEGIN
   IF (Dividend REM Divisor = 0) THEN
       Divides := TRUE
   ELSE Divides := FALSE
END;
```

Divides is a Boolean function that returns a Boolean value. If Divisor is a factor of Dividend (that is, it can be divided into Dividend with a zero remainder), the function Divides returns a value of TRUE. Otherwise, the function Divides returns a value of FALSE.

7.4 PARAMETERS

You pass data (that is, values and/or identifiers) to a subprogram by means of parameters. Parameters can be either value parameters or variable parameters. Value parameters are those in which the value is passed only to the subprogram; another value is not passed back from the subprogram. Variable parameters are those in which the value is passed to the subprogram and a changed value is passed back to the block from which the subprogram is called.

Formal parameters and actual parameters are terms used to describe the assignment process of the variables. Formal parameters are those listed in the subprogram declaration. Actual parameters are those specified in the subprogram call. In the body of the subprogram, the formal parameters represent the actual parameters.

7.4.1 Actual and Formal Parameters

A subprogram's formal parameters assume the values of the actual parameters when the subprogram is called. Formal parameters are listed in the heading of the subprogram. The formal parameters describe the parameters used within the subprogram and their types. The formal parameter list is the declaration for the parameters; they are not declared elsewhere. For example, the heading for the procedure Switch is shown below:

```
PROCEDURE Switch (VAR A, B :  INTEGER);
```

In the procedure, A and B are formal parameters, and they are both of the type INTEGER.

The format of the formal parameter list for specifying value or variable parameters is:

([[VAR]] identifier list : type [[; [[VAR]] identifier list : type...]])

The reserved word VAR is used to define variable parameters. Variable parameters return a changed value to the main body of the program. The identifier list specifies one or more variables that denote formal parameters. The type specifies the type of the parameters in the preceding identifier list.

The formal parameter list determines the nature of the actual parameter list. In the actual parameter list, you must include exactly one actual parameter for each formal parameter specified in the subprogram heading. All variables used in the actual parameter list must be declared or defined in the block surrounding the subprogram call.

For example, the following is a procedure heading with a formal parameter list:

```
PROCEDURE Change (New_Number:INTEGER;  VAR New_Letter:CHAR);
```

This is a valid procedure call to the procedure Change:

```
Change (Number, Letter);
```

The actual parameter Number is associated with the formal parameter New_Number and the actual parameter Letter is associated with the formal parameter New_Letter.

Thus, each actual parameter corresponds to a formal parameter. This correspondence is established solely on the basis of position in the respective parameter lists. The type of an actual parameter must be the same as that of its corresponding formal parameter.

You can call a subprogram several times with different actual parameters. For example, the same program that contains the procedure call to Change shown above can contain the following, if Old_Letter and Old_Number have been defined in the VAR section:

```
Change (Old_Number,Old_Letter);
```

As a result of this procedure call, when Change is called, the formal parameters New_Number and New_Letter assume the values of the actual parameters Old_Number and Old_Letter, respectively. See Section 7.4.3 for the program containing the procedure Change.

7.4.2 Value and Variable Parameters

You can specify two kinds of parameters in the formal parameter list of a subprogram: value parameters and variable parameters. The kind of parameter specified in the subprogram heading determines how the parameter is passed to the subprogram.

Value parameters pass the value of the actual parameter to the subprogram. The subprogram cannot change the actual parameter's value during execution. Variable parameters pass the address of the actual parameter variable to the subprogram, thus the subprogram can change the actual parameter's value.

PASCAL passes value parameters to subprograms by default. When you use the VAR specifier, the memory address of the actual parameter is passed to the subprogram. Therefore, the formal parameter and the actual parameter with which it is associated represent the same variable. When the subprogram changes the value of the formal parameter, it really changes the value of the actual parameter.

The following example shows the use of a program that uses value parameters with a function:

```
PROGRAM Dragon (INPUT,OUTPUT);

VAR
  Gold, Silver, Copper : INTEGER;

(* The function Wealth calculates the wealth of the player, based
on the gold, silver, and copper the player has. *)

FUNCTION Wealth (Gold_Coins,Silver_Coins,Copper_Coins : INTEGER) : REAL;

VAR
  Total_Wealth : REAL;

(* One gold coin is worth $25, one silver coin is worth $2.50,
and one copper coin is worth $.25 *)

BEGIN
        Total_Wealth := Gold_Coins * 25.0;
        Total_Wealth := ((Silver_Coins * 2.5) + Total_Wealth);
        Total_Wealth := ((Copper_Coins * 0.25) + Total_Wealth);
        Wealth := Total_Wealth;
END;
                (* Main program *)

BEGIN
        WRITELN ('How many coins does your player have?');
        WRITE ('Gold:');
        READLN (Gold);
        WRITE ('Silver:');
        READLN (Silver);
        WRITE ('Copper:');
        READLN (Copper);
        WRITELN ('Total wealth is: $',Wealth(Gold,Silver,Copper):8:2);
END.
```

In the main block of this program, values are entered at the terminal. These values are assigned with the READLN statement to the variables Gold, Silver, and Copper. These values are passed to the function Wealth with the call:

```
WRITELN ('Total wealth is:  $', Wealth (Gold, Silver, Copper):8:2);
```

This statement calls the function Wealth, using Gold, Silver, and Copper as value parameters. The function Wealth calculates the Total_Wealth, and then assigns that value to the function name. Control is returned to the main block, and the WRITELN statement is executed, using the newly calculated value of Wealth.

Because the values of Gold, Silver, and Copper are passed as value parameters, the values remain the same. They are not changed by execution of the Wealth subprogram.

The following is the output of the program Dragon, from compilation to execution:

```
@PASCAL
PASCAL>dragon.pas
PASCAL>/exit
@load dragon.rel
LINK:      Loading

EXIT
@save dragon.exe
 DRAGON.EXE.2 Saved
@run dragon
How many coins does your player have?
Gold:25
Silver:56
Copper:7
Total wealth is: $  766.75
@
```

The program Swap, shown at the beginning of the chapter, shows the use of variable parameters in a procedure.

```
PROCEDURE Switch (VAR A, B : INTEGER);

    VAR Temp: INTEGER;

    BEGIN
       Temp := B;
       B := A;
       A := Temp;
    END;
```

When the procedure Switch is called, the values of X and Y are assigned to A and B respectively. The procedure switches the values of A and B; and, since they are variable parameters, these changes in value also apply to the actual parameters X and Y.

Combining Value and Variable Parameters

The following example shows the use of both value and variable parameters in a function:

```
FUNCTION Symmetry (VAR SymArr : SquareArr;
                   Side : INTEGER) : BOOLEAN;

    VAR I,J : INTEGER;
    BEGIN
      Symmetry := TRUE;
      FOR I  := 1 TO Side DO
        FOR J := I TO Side DO
          IF A[I,J] <> A[J,I] THEN
             Symmetry := FALSE
    END;
```

The function Symmetry returns TRUE if a two-dimensional array is symmetric and FALSE if it is not symmetric. Suppose the type Square_Arr is defined as follows:

```
TYPE Square_Arr = ARRAY[1..10,1..10] OF INTEGER;
```

That is, it is a 100-element array of integers. The parameter Side is of type INTEGER and holds the length of the sides of the current array being tested.

The function heading specifies that an actual parameter of type Square_Arr will be passed as a variable parameter. However, the actual parameter of type INTEGER that is passed to Side is a value parameter.

You can reference the function Symmetry as follows:

```
IF Symmetry (This_Arr, This_Side) THEN
   WRITELN ('The array is Symmetric');
```

Say that This_Arr is of type Square_Arr, and This_Side is of type INTEGER. This_Arr will be passed as a variable parameter, and This_Side will be passed as a value parameter.

7.4.3 Examples

In the following examples, the program Pass demonstrates the use of these terms. The procedure Change is used to show the use of local variables, global variables, value parameters, and variable parameters.

```
PROGRAM Pass (INPUT,OUTPUT);

VAR
        Number : INTEGER;
        Letter : CHAR;
        Limit : REAL;

PROCEDURE Change (New_Number : Integer; VAR New_Letter : CHAR);
VAR
        I : INTEGER;

        (* statements for the procedure Change*)

BEGIN
    FOR I := 1 TO 3 DO
        BEGIN
            WRITELN ('Enter a new value for Number:');
            READLN (New_Number);
            WRITELN ('Enter a new value for Letter:');
            READLN (New_Letter);
            WRITELN ('Enter a new value for Limit:');
            READLN (Limit);
        END
END;

                    (* main body of the program *)

BEGIN
    Number := 5;
    Letter := 'A';
    Limit := 12.5;
    WRITELN ('The value of Number is:',Number:3);
    WRITELN ('The value of Letter is:',Letter:2);
    WRITELN ('The value of Limit is:', Limit:4:2);
    Change (Number, Letter);
    WRITELN ('The new value of Number is:',Number:3);
    WRITELN ('The new value of Letter is:',Letter:2);
    WRITELN ('The new value of Limit is:', Limit:4:2);
END.
```

Example 1

The following declaration shows the definition of global variables for the program:

```
VAR
    Number : INTEGER;
    Letter : CHAR;
    Limit : REAL;
```

These global variables are defined in the declaration section associated with the main block of the program; these variable names can be used anywhere in the main block or in any of the subprograms defined in the declaration section.

Example 2

The following shows the use of formal parameters in the procedure heading:

```
(New_Number : INTEGER; VAR New_Letter : CHAR);
```

The formal parameter list lists the variable names that are associated with variable names used in the procedure call. The names of the formal parameters are unknown outside the subprogram, and therefore cannot be used elsewhere in the program.

Example 3

The following shows the use of a value parameter:

```
New_Number : INTEGER;
```

The variable name New_Number is associated (when the procedure is called) with a variable name in the procedure call.

A value parameter does not pass a value from the subprogram back to the block from which it is called. The type must be specified, and the type must agree with the type of the variable named in the procedure call.

Example 4

The following shows the use of a variable parameter:

```
VAR New_Letter : CHAR
```

The variable name New_Letter is associated (when the procedure is called) with a variable name in the procedure call.

A variable parameter passes the value from the subprogram back to the block from which it is called. Variable parameters must be preceded with the reserved word VAR. As with value parameters, the type must be specified, and the type must agree with the type of the variable named in the procedure call.

Example 5

This example contains a local variable. A local variable can be used only in the subprogram in which it is defined. It does not pass any values to or from the subprogram. Local variables are frequently used in FOR statements in subprograms.

```
PROCEDURE Change (New_Number : INTEGER; VAR New_Letter : CHAR);

VAR
     I : INTEGER;
       .
       .
       .
```

Local variables in subprograms are defined exactly as if they were being defined in the declaration section for the main block of the program.

Example 6

This is the call to the subprogram:

```
BEGIN
    Change (Number, Letter);
    .
    .
    .
```

The call includes the procedure name and the actual parameter list. The actual parameter list includes the names of all the variables that are being used to pass data to the subprogram. The actual parameter list must coincide with the formal parameter list in two ways: the number of variables must match, and the type of variables must match.

When a procedure is called, the leftmost variable in the actual parameter list is associated with the leftmost variable in the formal parameter list.

```
(New_Number : INTEGER; VAR New_Letter : CHAR);

Change (Number, Letter);
```

The top line shows the formal parameter list. The second line shows the actual parameter list. When the procedure is called, New_Number is associated with Number, and New_Letter is associated with Letter. When the flow of the program returns to the main block, the value associated with New_Letter is assigned to Letter. The value associated with New_Number is not passed out of the procedure because New_Number is a value parameter, and values of value parameters are not returned from the subprogram.

APPENDIX A

PASCAL DEFINED NAMES

This appendix contains lists of the names defined by PASCAL. Section A.1 lists the reserved words that cannot be redefined as identifiers. Section A.2 lists the semireserved words that can be redefined. Section A.3 lists the predeclared identifiers that hold a special meaning to PASCAL, but can, if necessary, be redefined as user identifiers.

A.1 RESERVED WORDS

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

A.2 SEMIRESERVED WORDS

MODULE
OTHERWISE
REM
VALUE

A.3 PREDECLARED IDENTIFIERS

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

APPENDIX B

ASCII CHARACTER SET

Table B-1 summarizes the ASCII character set. Each element of the ASCII character set is a constant value of the PASCAL predefined type CHAR. The ASCII decimal number in Table B-1 is the same as the ordinal value (as returned by the PASCAL ORD function) of the associated character in the type CHAR.

Table B-1: The ASCII Character Set

ORD Decimal Number	Character	Meaning
0	NUL	Null
1	SOH	Start of heading
2	STX	Start 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

Table B-2: The ASCII Character Set (Cont.)

ORD Decimal Number	Character	Meaning
35	#	Number sign
36	$	Dollar sign
37	%	Percent sign
38	&	Ampersand
39	'	Apostrophe
40	(	Left parenthesis
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	Uppercase A
66	B	Uppercase B
67	C	Uppercase C
68	D	Uppercase D
69	E	Uppercase E
70	F	Uppercase F
71	G	Uppercase G
72	H	Uppercase H
73	I	Uppercase I
74	J	Uppercase J
75	K	Uppercase K
76	L	Uppercase L
77	M	Uppercase M
78	N	Uppercase N
79	O	Uppercase O
80	P	Uppercase P
81	Q	Uppercase Q
82	R	Uppercase R
83	S	Uppercase S
84	T	Uppercase T
85	U	Uppercase U
86	V	Uppercase V
87	W	Uppercase W
88	X	Uppercase X
89	Y	Uppercase Y
90	Z	Uppercase Z
91	[	Left square bracket

Table B-2: The ASCII Character Set (Cont.)

ORD Decimal Number	Character	Meaning
92	\	Back slash
93	]	Right square bracket
94	^ or	Circumflex or up arrow
95	or _	Back arrow or underscore
96	`	Grave accent
97	a	Lowercase a
98	b	Lowercase b
99	c	Lowercase c
100	d	Lowercase d
101	e	Lowercase e
102	f	Lowercase f
103	g	Lowercase g
104	h	Lowercase h
105	i	Lowercase i
106	j	Lowercase j
107	k	Lowercase k
108	l	Lowercase l
109	m	Lowercase m
110	n	Lowercase n
111	o	Lowercase o
112	p	Lowercase p
113	q	Lowercase q
114	r	Lowercase r
115	s	Lowercase s
116	t	Lowercase t
117	u	Lowercase u
118	v	Lowercase v
119	w	Lowercase w
120	x	Lowercase x
121	y	Lowercase y
122	z	Lowercase z
123	{	Left brace
124	|	Vertical line
125	}	Right brace
126	~	Tilde
127	DEL	Delete

APPENDIX C

SUMMARY OF PREDECLARED PROCEDURES AND FUNCTIONS

Tables C-1 and C-2 summarize the procedures and functions declared by PASCAL. Some of the listed procedures and functions are not described in this primer. For information on these, see the TOPS-20 PASCAL Language Manual.

Table C-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	Release storage for p^. The pointer variable p becomes undefined.
DISPOSE(p, t1,...,tn)	p = pointer variable t1,...,tn = tag field constants	Release storage for 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.

Table C-1: Predeclared Procedures (Cont.)

Procedure	Parameter Type	Action
HALT	None	Calls LIB$STOP, which signals SS$_ABORT. Without an appropriate condition handler, HALT 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.
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 the TOPS-20 PASCAL Language Manual	Opens the file f with the specified attributes.

Table C-1: Predeclared Procedures (Cont.)

Procedure	Parameter Type	Action
PACK(a,i,z)	a = variable of type ARRAY [m..n] OF T i = starting subscript 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[i] through a[i+v-u] to z[u] through z[v]. The upper bound of a subscript must be greater than or equal to (i+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.
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.
READ(f, v1,...,vn)	f = file variable v1,...,vn = variables	Assigns successive values from the input file f to the variables v1 through vn. 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, then sets the current file position to the beginning of the next line. The variables v1 through vn are optional. The default for f is INPUT.
RELEASE(a)	a = a variable of type ^INTEGER	Deallocates memory allocated by the NEW procedure up to the marker set by the MARK procedure.

Table C-1: Predeclared Procedures (Cont.)

Procedure	Parameter Type	Action
RESET(f)	f = file variable	Enables reading from file f. RESET(f) moves the current file position to the beginning of the 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) truncates 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 subscript 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[1+v-u]. The upper bound a subscript must be greater than or equal to (1+v-u).
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.

Table C-2: Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
Arithmetic	ABS(x)	Integer, real, double	Same as x	Computes the absolute value of x.
	ARCTAN(x)	Integer, real, Double	Real Double	Computes the arctangent of x.
	COS(x)	Integer, real Double	Real Double	Computes the cosine of x.
	EXP(x)	Integer, real Double	Real Double	Computes e**x, the exponential function.
	LN(x)	Integer, real Double	Real Double	Computes the natural logarithm of x. The value of x must be greater than 0.
	SIN(x)	Integer, real Double	Real Double	Computes the sine of x.
	SQR(x)	Integer, real, double	Same as x	Computes x**2, the square of x.
	SQRT(x)	Integer, real Double	Real Double	Computes the square root of x. If x is less than zero, PASCAL generates an error.

Table C-2: Predeclared Functions (Cont.)

Category	Function	Parameter Type	Result Type	Purpose
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 component 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 position pointer is after the last character in a line, in which case the value 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 is the value of r is not in the proper floating-point format.

Table C-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 (if one exists) whose ordinal value is n.
	ORD(x)	Any ordinal type	Integer	Returns the ordinal value corresponding to the value of x.
	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.

Table C-2: Predeclared Functions (Cont.)

Category	Function	Parameter Type	Result Type	Purpose
Misc	CLOCK	None	Integer	Returns an integer value equal to the central processor time used by the current process. The time is expressed in milliseconds.
	EXPO(r)	Real, double	Integer	Returns the integer-valued exponent of the floating-point representation of r.
	PRED(x)	Any ordinal type	Same as x	Returns the predecessor value in the type of x (if a predecessor exists).
	SUCC(x)	Any ordinal type	Same as x	Returns the successor value in the type of x (if a successor exists).

GLOSSARY

actual parameter

Value or variable that is passed in a procedure or function call and is used during execution of the subprogram.

actual parameter list

List of actual parameters specified in a subprogram call. The actual parameters must be separated by commas, and the entire list must be enclosed in parentheses.

address

Specification of a location in the computer's memory.

arithmetic expression

Expression that evaluates to a real or integer value.

arithmetic operator

Symbol used with numeric variables, function references, and constants in forming arithmetic expressions. In PASCAL, the arithmetic operators are +, -, *, /, **, DIV, MOD, and REM.

array

Collection of data items, called components, that have the same type and share an identifier. The components can be accessed by the array identifier and an index enclosed in brackets. See also index.

ASCII character set

Characters and output control data that represent characters in PASCAL. (ASCII stands for American Standard Code for Information Interchange.) Each member of the ASCII character set corresponds to a unique integer between 0 and 127, inclusive.

assignment statement

Executable statement that assigns a value to a variable.

base type

(1) For a subrange, the scalar type of which it is a subset. For example, the subrange 0..123 has the base type INTEGER. (2) For a set, the nonreal scalar type from which the elements of the set are chosen.

block

Declaration and executable sections of a program, procedure, or function. One block can be nested in another; for example, a procedure block is nested in the block of its declaring program.

Boolean expression

Expression that evaluates to one of the Boolean values, FALSE or TRUE.

BOOLEAN type

Predefined scalar type that has the identifiers FALSE and TRUE as constant values.

bound

Upper or lower limit of a subrange, often used in defining the index limits for a dimension of an array.

case label

In the CASE statement, a constant of the same type as the case selector. A list of case labels, separated by commas and followed by a colon (:), precedes each statement that can be chosen for execution.

case selector

In the CASE statement, the expression whose value determines the statement selected for execution. In choosing the statement to be executed, the CASE statement first evaluates the case selector. It then scans the case labels preceding each possible statement, and executes the statement that is labeled with the value of the case selector.

character

Single element of the ASCII character set and a value of the predefined type CHAR.

character string

Sequence of ASCII characters, enclosed in apostrophes. See also string constant and string variable.

CHAR type

Predefined scalar type that has the ASCII character set as constant values.

comment

Any sequence of characters appearing between the character pairs (* and *) or { and }. Comments are for documentation purposes and are ignored by PASCAL.

compiler

Program that translates source program statements into an object module.

component

In an array, an individual data item denoted by the array name and an index for each dimension.

compound statement

One or more PASCAL statements, bracketed by the reserved words BEGIN and END, that are executed sequentially as a unit.

conditional statement

Statement that selects another statement for execution, depending on the value of an expression. PASCAL's conditional statements are IF-THEN, IF-THEN-ELSE, and CASE.

constant

Literal that represents a value that cannot change during program execution. Examples include 'p', 3, 2.781, and 'metaphysics'.

control statement

Statement that directs the flow of control in a program, such as the IF-THEN-ELSE, FOR, WHILE, or CASE statement.

control variable

Scalar variable that takes on sequential values with each iteration of a FOR loop. After completion of the loop, the value of the control variable is undefined.

data structures

Combinations of single data items that form related groups of data, for example, records or arrays.

data type

See type.

decimal notation

Representation of real numbers in the integer.fraction format, as in 23.27, -7.83, and 0.0.

declaration

Specification that lists one or more labels or that associates an identifier with what it represents. Labels and identifiers for constants, types, variables, procedures, and functions must be declared.

declaration section

The part of a program or subprogram block that contains the declaration and definitions.

default

Action taken or value assumed by the system when none is explicitly specified.

delimiter

Punctuation mark or reserved word that sets off one part of a PASCAL program from another. BEGIN and END enclose the executable section or a compound statement. The semicolon (;) separates declarations or statements, and the period (.) indicates the end of the program.

dimension

Range of values for one subscript of an array. An array can have any number of dimensions; see multidimensional array.

double precision

Precision of approximately 16 significant digits for a floating-point real number; the type DOUBLE provides double precision.

DOUBLE type

Predefined scalar type that has double-precision real numbers as values.

end-of-file

Condition indicating that the current file contains no more data. The EOF function tests this condition.

end-of-line

Condition indicating that the current line contains no more data. The EOLN function tests this condition.

enumerated type

Type comprising a sequence of values denoted by identifiers. A list of identifiers, separated by commas and enclosed in parentheses, defines an enumerated type.

executable image

File containing the executable version of a program. An executable image is the output from the linker, and is created by linking one or more object modules.

executable section

The part of a block that contains the executable statements, delimited by BEGIN and END, that perform the actions of the block.

executable statement

Statement in the executable section of the block that performs an action or directs the flow of control.

expression

A constant, a variable, a function reference, or some combination of constants, variables, function references, operators, and parentheses that PASCAL can evaluate. Every expression is associated with a type.

external file

File that exists outside the scope of the PASCAL program, and therefore must be specified in the program heading.

field

Named component of a record, containing data items of one or more types. References to a field are in the record.fieldname format.

field width

Minimum number of characters that the WRITE or WRITELN procedure writes to a text file for a particular expression or character string.

file

See file variable.

file component

Accessible unit of a file variable. A file component can be of any type except a file type or a structured type containing elements of a file type.

file name

On TOPS-20, a 1- to 39- character name component of a source file specification. However, the name of a relocatable binary file must contain no more than six characters.

file generation number

Numeric component of a TOPS-20 file specification. As a file is updated and changed, the file's generation number is updated with each successive copy.

file position

Position immediately following the file component that was last read or written. Only the component at the current file position can be accessed.

file specification

Unique TOPS-20 identification of a file on a mass storage medium (such as a disk). It describes the physical location of the file (node, device, and directory) and identifies the file name, file type, and file generation number.

file type

(1) On TOPS-20, in the source file specification, the 0- to 39-character type component that usually describes the nature of a file or how it is used. For example, PAS indicates a PASCAL source program. However, the file type of the rel file must contain no more than three characters. (2) In PASCAL, a structured type that is a sequence of any number of data components of the same type.

file variable

Named sequence of components of the same type used in I/O operations. A file can have any number of components; they can be of any type except a file type or a structured type containing components of a file type.

floating-point notation

Representation of real number data in the mantissa.fraction format, followed by a positive or negative exponent. The exponent is introduced by the letter E for a single-precision number, such as 7.321E02, and the letter D for a double-precision number, such as 4.0D-6.

formal parameter

Name that is declared in the heading of a procedure or function, and that represents an actual parameter when the procedure or function is invoked.

formal parameter list

List of passing mechanisms and formal parameter declarations that appears in the heading of a procedure or function. The entries in the list are separated by semicolons, and the list is enclosed in parentheses.

function

Program unit that returns a value when executed. A function consists of a heading, which includes the function's name and result type, and a block.

function heading

Specification of the name, formal parameter list, and result type of a function in a function declaration.

function reference

Use of a function name and actual parameter list in an executable statement to invoke the function.

global identifier

Identifier that is declared in a block at an outer level and therefore can be used inside the current (inner-level) block without redeclaration.

global variable

Variables that are declared in the declaration section for the main block of the program. Their scope is the entire program.

heading

Specification that precedes a block and defines the block's name and parameters.

identifier

One or more alphanumeric characters that denote a variable, constant, type, procedure, function, or other item that is not a reserved word. Although an identifier can be any length, PASCAL treats only the first 31 characters as significant.

index

An expression of an ordinal type that is used with an array name to specify a component of that array.

input procedure

Procedure that reads data into a program. PASCAL provides three predeclared input procedures: READ, READLN, and GET.

integer

In PASCAL, a whole number between -34359738368 and +34359738367; a value of type INTEGER.

INTEGER type

Predefined scalar type that has integers as values.

interactive mode

Mode of communication in which the system responds to the commands and program input that the user types at the terminal.

internal file

File that exists only within the scope of a block and is deleted when execution of that block terminates.

label

(1) See case label. (2) Unsigned integer constant that is declared in the LABEL section and used to make a statement accessible from a GOTO statement. The label precedes the statement and is separated from it by a colon (:).

LINK

Program that creates an executable image from one or more object modules. Programs must be linked before they can be executed.

LOAD

Command used to invoke the LINK program.

local identifier

Identifier that is declared within a block and is unknown, and therefore inaccessible, outside that block.

logical operator

Reserved word or symbol that specifies a logical test. The logical operators in PASCAL include AND, OR, and NOT.

loop body

One or more statements that are executed repetitively until a specified condition is met.

multidimensional array

Array with elements of an array type. Each dimension of a multidimensional array has its own subscripts, which can be of different types.

modulus

Remainder of dividing one number by another. The MOD and the REM functions return the modulus.

nested

Contained within, as in a function declared within a procedure or a record that is a field of another record.

object module

Binary output from a language compiler or assembler that is input to the linker. The linker processes one or more object modules to produce an executable image.

operator

Symbol used in an expression to cause PASCAL to perform a specific task. PASCAL includes arithmetic, relational, and logical operators.

ordinal

Term used to encompass all nonreal scalar data types; ordinal refers to integer, character, and Boolean data types.

ordinal value

Integer corresponding to the position of a given value in a sequential list of values of its type. Ordinal value applies only to integer, character, Boolean, enumerated, and subrange types. The ORD function returns the ordinal value of an expression of one of these types.

output procedure

Procedure that writes data into a file. PASCAL provides three predeclared output procedures: WRITE, WRITELN, and PUT.

packed

Stored as densely as possible in the computer's memory.

parameter

Means of passing information between program units. See actual parameter, formal parameter, read parameters, and write parameters.

parameter list

Specification of the actual or formal parameters for a procedure or function. The parameter list follows the name of the procedure or function and is enclosed in parentheses. See also actual parameter list and formal parameter list.

Pascal

(1) Blaise Pascal, a French mathematician and philosopher, born in 1623 and died in 1662. (2) Structured programming language developed by Niklaus Wirth in Zurich, Switzerland in the early 1970s.

pointer

Variable whose value is a reference to a dynamic variable.

precedence rules

Rules applied to the order of evaluation of operations in an expression. An operation with higher precedence is performed before an operation with lower precedence.

predecessor value

Value that immediately precedes a given value in any nonreal scalar type. The PRED function returns the predecessor value.

predeclared

Declared by PASCAL rather than by the programmer.

predeclared identifier

Identifier declared by PASCAL to name a type, constant, variable, procedure, or function.

predeclared subprogram

Procedure or function declared by PASCAL and available for use without further declaration.

predefined

See predeclared.

procedure

Program unit that consists of a procedure heading and a block. When called, a procedure is executed as a unit. See also predeclared subprogram.

procedure call

Statement that invokes a procedure. A procedure call consists of the name of a procedure and its actual parameter list (when required).

procedure heading

Specification of the name and optional formal parameters of a procedure in a procedure declaration.

program heading

Specification that begins a PASCAL program. The program heading specifies the program's name and its external files.

read parameters

Variables used as parameters in a call to the READ or READLN procedure to which input values will be assigned.

real number

In PASCAL, the floating-point internal representation of a number that can be of any size; however, each number can be rounded to fit the precision of 27 bits (7 to 9 decimal digits). The value 0.0 is also included.

REAL type

Predefined scalar type that has single-precision real numbers as values; synonymous with SINGLE type.

record

Organized collection of data containing one or more fields, each of which can be of a different type.

relational operator

Symbol that tests the relationship between two values, the result of which is one of the Boolean values, FALSE or TRUE. PASCAL's relational operators are <, >, <=, >=, and <>.

repetitive statement

Statement that causes an action to be performed iteratively. PASCAL's repetitive statements are FOR, REPEAT, and WHILE.

reserved word

Word set aside by the PASCAL compiler as the name for a declaration, statement, data structure, delimiter, or operator. Reserved words have special meanings to the compiler and cannot be used as identifiers.

return value

Result of a function, assigned to the function's name during its execution. The return value is supplied to the calling program wherever a reference to the function appears.

scalar data type

Type in which the values are unique and indivisible units of data. The values of a scalar type follow a particular order. Predefined scalar types include INTEGER, REAL, CHAR, and BOOLEAN.

scope

Portion of the program in which an identifier has a particular meaning. The scope of an identifier is the block in which it is declared.

semireserved word

Word set aside by the PASCAL compiler that has a special meaning to the compiler. Semireserved words can be used as identifiers.

set

Collection of nonreal scalar elements, called members.

single precision

Precision of approximately seven significant digits for a floating-point real number; the types SINGLE and REAL provide single precision.

SINGLE type

Predefined scalar type that has single-precision real numbers as values; synonymous with REAL type.

source file

TOPS-10 or TOPS-20 file that contains source program statements used as input to a language compiler.

statement

Sequence of reserved words, identifiers, operators, expressions, and special symbols describing a program action or altering the flow of program execution.

string

See character string.

string constant

Character string used as a literal constant in the program, for example 'one pink rose'.

string variable

Variable of type PACKED ARRAY [1..n] OF CHAR, where n represents an integer constant.

structured data types

Collection of related data components. The components can be of the same type (as for arrays and files) or of different types (as for records).

subprogram

Procedure or functions; used in this manual in descriptions that apply to both procedures and functions.

subrange types

Subset of an existing scalar type, defined for use as a type. A subrange must be a continuous range of values, and is described by its upper and lower bounds separated by the .. symbol.

successor value

Value that succeeds a given value in any nonreal scalar type. The SUCC function returns the successor value.

symbolic constant

Name defined to represent a constant value; can be used in place of the value.

symbolic name

Word used in a PASCAL program. A symbolic name can be a reserved word, a predeclared identifier, or a user identifier.

text file

File that has components of type CHAR and is implicitly divided into lines.

type

Set of values, usually named with an identifier, for which certain operations are defined. Some types are defined by PASCAL -- INTEGER, REAL, SINGLE, DOUBLE, BOOLEAN, and CHAR. Others are defined by the programmer.

user-defined type

Type defined by the programmer. User-defined types can be scalar (enumerated or subrange), structured, or pointer.

user identifier

Identifier created by the programmer to denote a program, constant, type, variable, procedure, or function.

value initialization

TOPS-20 extension that allows a programmer to assign a constant value to a variable in the program's declaration section.

value parameters

Parameters in which the value is passed only to the subprogram. Another value is not passed back from the subprogram.

variable

Data item (of fixed type) that can change in value during execution of the program.

variable parameters

Parameters in which the value is passed to the subprogram, and a changed value is passed back to the block from which the subprogram is called.

write parameters

Expressions that are specified as parameters to the WRITE or WRITELN procedure, which writes them in the specified file.

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