VAX–11 PASCAL Primer

Order No. AA-J180B-TE

October 1982

This tutorial document introduces the VAX–11 PASCAL language. It is intended to be used by programmers who are new to VAX–11 PASCAL.

REVISION/UPDATE INFORMATION:	This revised document supersedes the *VAX–11 PASCAL Primer* (Order No. AA-J180A-TE).
OPERATING SYSTEM AND VERSION:	VAX/VMS V3.1
SOFTWARE VERSION:	VAX–11 PASCAL V2.0

digital equipment corporation · maynard, massachusetts

First Printing, April 1980
Revised, October 1982

DEC	DIBOL	RSX
DEC/CMS	Edusystem	UNIBUS
DECnet	IAS	VAX
DECsystem-10	MASSBUS	VMS
DECSYSTEM-20	PDP	VT
DECUS	PDT	digital
DECwriter	RSTS	

ZK2096

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

Contents

Page

Preface vii

Chapter 1 Introduction

1.1 Program Development . . . 1–2

1.1.1 Creating the Program . . . 1–4
1.1.2 Compiling the Program . . . 1–4
1.1.3 Linking the Object Module . . . 1–5
1.1.4 Executing the Program . . . 1–5

1.2 A PASCAL Program Example . . . 1–6
1.3 The Structure of a PASCAL Program . . . 1–6

1.3.1 The Program Heading . . . 1–8
1.3.2 The Declaration Section . . . 1–9
1.3.3 The Executable Section . . . 1–10

Chapter 2 Data Concepts

2.1 Types . . . 2–1
2.2 Scalar Types . . . 2–2

2.2.1 The Type INTEGER . . . 2–3
2.2.2 The Type REAL . . . 2–3
2.2.2.1 Decimal Notation . . . 2–3
2.2.2.2 Floating-Point Notation . . . 2–4
2.2.3 The Type BOOLEAN . . . 2–4
2.2.4 The Type CHAR . . . 2–5

2.3 Variables . . . 2–5
2.4 Expressions . . . 2–6

2.4.1 Arithmetic Expressions . . . 2–6
2.4.2 Relational Expressions . . . 2–9
2.4.3 Logical Expressions . . . 2–10
2.4.4 Precedence Rules for Operators . . . 2–11

Chapter 3 Declarations and Definitions

3.1 Symbolic Names . . . 3–1

3.1.1 Reserved Words and Predeclared Identifiers . . . 3–2
3.1.1.1 Reserved Words . . . 3–2
3.1.1.2 Predeclared Identifiers . . . 3–2
3.1.2 User Identifiers . . . 3–3

3.2 Constant Definitions . . . 3–4
3.3 Type Definitions . . . 3–5
3.4 Variable Declarations . . . 3–6
3.5 User-Defined Scalar Types . . . 3–7

3.5.1 Enumerated Types . . . 3–8
3.5.2 Subrange Types . . . 3–10

Chapter 4 Fundamental PASCAL Statements

4.1 The Assignment Statement 4–1
4.2 The Compound Statement 4–3
4.3 The IF-THEN Statement 4–3
4.4 The IF-THEN-ELSE Statement 4–5
4.5 The FOR Statement 4–8

Chapter 5 Reading and Writing Data

5.1 The Predeclared Text Files Input and Output 5–1
5.2 Reading Data 5–2
5.2.1 The READ Procedure 5–2
5.2.2 The READLN Procedure 5–4
5.3 Writing Data 5–5
5.3.1 The WRITE Procedure 5–5
5.3.2 The WRITELN Procedure 5–9
5.4 The Predeclared Functions EOLN and EOF 5–10
5.4.1 The EOLN Function 5–10
5.4.2 The EOF Function 5–11

Chapter 6 Structured Types: the Array and the Record

6.1 Arrays 6–1
6.1.1 Multidimensional Arrays 6–5
6.1.2 Character Strings 6–8
6.1.2.1 Character-String Constants 6–8
6.1.2.2 Character-String Variables 6–9
6.2 Records 6–11

Chapter 7 More PASCAL Statements

7.1 The REPEAT Statement 7–1
7.2 The WHILE Statement 7–5
7.3 The CASE Statement 7–7
7.4 The Program Class_Data — An Example 7–10
7.4.1 The Declaration Section 7–12
7.4.2 The Executable Section 7–12

Chapter 8 Procedures and Functions

8.1 Procedures 8–6
8.1.1 Declaring a Procedure 8–6
8.1.2 Calling a Procedure 8–6
8.2 Functions 8–7
8.2.1 Declaring a Function 8–7
8.2.2 Invoking a Function 8–8

8.3 Parameters 8–8
8.3.1 Actual and Formal Parameters 8–9
8.3.2 Value and Variable Parameters 8–10
8.3.2.1 Value Parameters 8–10
8.3.2.2 Variable Parameters 8–11

Appendix A PASCAL Defined Names

A.1 Standard Reserved Words A–1
A.2 Nonstandard Reserved Words. A–1
A.3 Predeclared Identifiers A–2

Appendix B ASCII Character Set

Appendix C Summary of Predeclared Procedures and Functions

Glossary

Index

Figures

1–1 Program Development Process 1–3
1–2 Sample Program Grocery_Bill 1–7
1–3 Executable Section of Grocery_Bill 1–10
1–4 Sample Run of Grocery_Bill. 1–12
4–1 The IF-THEN Statement Flow Chart. 4–4
4–2 The IF-THEN-ELSE Statement Flow Chart. 4–6
4–3 The FOR Statement Flow Charts 4–9
5–1 The End of a Text File. 5–12
5–2 File Position at End-of-File. 5–12
6–1 The Two-Dimensional Array Class_Scores 6–6
6–2 The Three-Dimensional Array Hotel_Vacancies. 6–8
7–1 The REPEAT Statement Flow Chart 7–2
7–2 The WHILE Statement Flow Chart. 7–5
7–3 The CASE Statement Flow Charts 7–9
7–4 The Program Class_Data 7–10
8–1 The Program Compute. 8–2

Tables

2–1 Arithmetic Operators. 2–7
2–2 Result Types for Arithmetic Expressions 2–8
2–3 Relational Operators 2–9
2–4 Logical Operators 2–10
2–5 Precedence of Operators 2–11
5–1 Default Values for Field Width 5–6
B–1 The ASCII Character Set. B–1
C–1 Predeclared Procedures. C–2
C–2 Predeclared Functions C–5

Preface

Primer Objectives

This primer introduces the VAX-11 PASCAL language. It is designed to provide sufficient information on the language for you to begin writing PASCAL programs.

VAX-11 PASCAL is an extended implementation of the PASCAL language that accepts programs which comply with the standard proposed by the International Organization for Standardization. This primer describes a subset of VAX-11 PASCAL, omitting some advanced features of the language. Once you have mastered the concepts in this primer, you should consult the *VAX-11 PASCAL Language Reference Manual* for full reference information.

Intended Audience

This primer does not attempt to teach programming concepts. It assumes that you have some 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 VAX/VMS operating system, but some familiarity with VAX/VMS is helpful. If you are new to VAX/VMS, see the *VAX/VMS Primer* for introductory material.

How to Use This Document

This primer contains eight chapters. Chapter 1 explains how to develop PASCAL programs on VAX/VMS. Program examples are found throughout; using the tools for program development introduced in Chapter 1, you can enter, compile, link, and run these sample programs on VAX/VMS. Each subsequent chapter introduces new concepts in PASCAL that build on material presented previously. To take advantage of this structure, you should read the chapters sequentially. Throughout the primer, italics indicate a term defined in the Glossary.

For More Information

For reference information on the VAX-11 PASCAL language, consult the *VAX-11 PASCAL Language Reference Manual.*

The *VAX-11 PASCAL User's Guide* provides information on using PASCAL with the VAX/VMS operating system.

The *VAX-11 Information Directory and Index* briefly describes each manual in the VAX/VMS document set. The information directory indicates which manuals you should consult for information on various components of the operating system.

Conventions Used in This Document

This document uses the following conventions.

Convention	**Meaning**
{ }	Braces enclose lists from which you must choose one item; for example: $\left\{ \begin{array}{c} \text{TO} \\ \text{DOWNTO} \end{array} \right\}$
...	A horizontal ellipsis means that the preceding item can be repeated one or more times; for example: digit...
{ },...	Braces followed by a comma and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating the items with commas; for example: {label},...
{ };...	Braces followed by a semicolon and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating the items with semicolons; for example: REPEAT {statement};... UNTIL expression
. . .	A vertical ellipsis means that not all of the statements in a figure or example are shown.

[ ]	Square brackets mean that the syntax requires the square bracket characters. This notation is used with arrays, sets, and attribute lists; for example: ARRAY [index]
⟦ ⟧	Double brackets enclose items that are optional; for example: EOLN ⟦ (file-variable) ⟧
`BEGIN`	In programming examples, all PASCAL names, that is, reserved words and predeclared identifiers, are printed in uppercase letters.
`Grocery_Bill`	In programming examples, all user identifiers, that is, names created by the programmer, are printed in lowercase letters with initial uppercase letters.
CTRL/X	The notation CTRL/X indicates that you must press the key labeled CTRL while simultaneously pressing another key, such as X in this example.
RET	A symbol with a 1- to 3-character abbreviation indicates a key that you press on the terminal; for example, RET for the RETURN key.
`$ RUN EXAMPLE`	Interactive examples are shown in two colors. User-entered input is printed in red. Program- and system-generated output is printed in black.
A PASCAL program consists of a *heading* and a *block.*	A word or phrase in italics indicates a term defined in the Glossary.

Chapter 1
Introduction

The *PASCAL* language was designed for teaching structured programming techniques; as such, it is used widely in educational institutions. It has also gained popularity as a general-purpose language because it is suitable for many different programming applications.

PASCAL programs are structured; that is, they use English-like statements that allow the programmer to make the logical flow efficient, readily discernible, and as linear as possible. As a result, PASCAL programs are easy to read, modify, and maintain. The names of all data items must be explicitly declared at the beginning of the program. The declaration section makes readily apparent the symbolic names, or identifiers, that represent constants, data types, variables, procedures, and functions.

The structure of a PASCAL program and the wide range of available data structures encourage modular programming. In modular programming, you divide the solution to a problem into individual parts that can be developed relatively independently. PASCAL's block structure promotes the translation of these parts into procedures and functions.

PASCAL includes a variety of control statements, data types, and predeclared procedures and functions. Some of the language features that are common to most implementations of PASCAL are:

- INTEGER, REAL, CHAR, BOOLEAN, enumerated, and subrange scalar data types
- ARRAY, RECORD, SET, and FILE structured data types
- FOR, REPEAT, and WHILE repetitive control statements
- CASE, IF-THEN, and IF-THEN-ELSE conditional statements
- BEGIN...END compound statement
- READ, WRITE, READLN, and WRITELN input and output procedures
- Standard set of predeclared functions and procedures

In addition to common PASCAL language features, this primer also presents the following VAX–11 *extensions* to the PASCAL language:

- Exponentiation operator
- Double- and quadruple-precision real data types
- 31-character identifiers that can include the dollar sign ($) and underscore (_) characters
- 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
- Initialization of variables
- Expressions in the CONST section

This chapter introduces the VAX–11 PASCAL language. Section 1.1 presents the steps required for developing a VAX–11 PASCAL program on the VAX/VMS operating system. By following these steps, you can run the programs that you will find throughout this primer. Section 1.2 presents a PASCAL program example. Section 1.3 uses this example to illustrate some of the fundamental concepts of PASCAL programs.

For more information on all aspects of the VAX–11 PASCAL language, see the *VAX–11 PASCAL Language Reference Manual.*

1.1 Program Development

This section explains the steps required for developing a VAX–11 PASCAL program. Figure 1–1 illustrates the program development process. Developing a VAX–11 PASCAL program involves four steps:

- Creating a *source file* containing the program source statements
- Compiling the source program to create an *object module*
- Linking the object module to produce an *executable image*
- Executing the image

You specify these steps by entering the following commands to the VAX/VMS operating system:

```
$ EDIT file-spec
$ PASCAL file-spec
$ LINK file-spec
$ RUN file-spec
```

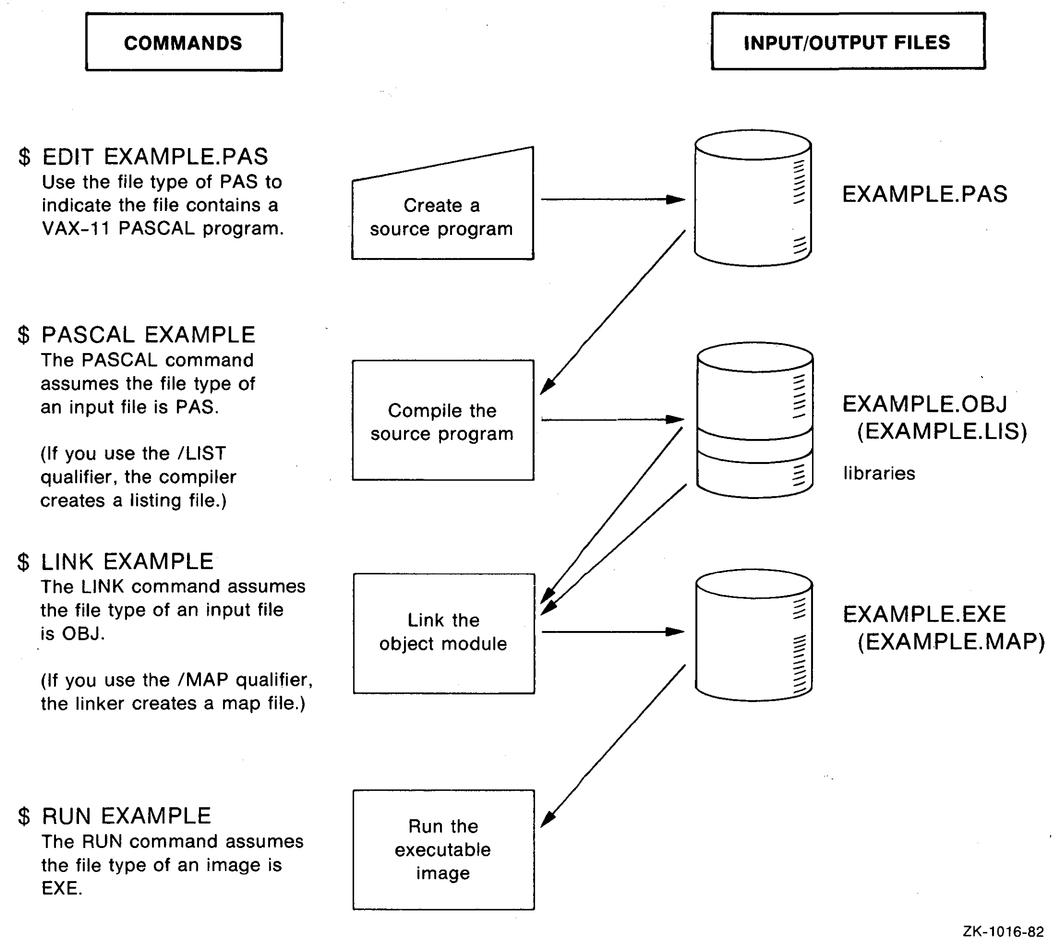
COMMANDS
INPUT/OUTPUT FILES
$ EDIT EXAMPLE.PAS
Use the file type of PAS to indicate the file contains a VAX-11 PASCAL program.
Create a source program
EXAMPLE.PAS
$ PASCAL EXAMPLE
The PASCAL command assumes the file type of an input file is PAS.
(If you use the /LIST qualifier, the compiler creates a listing file.)
Compile the source program
EXAMPLE.OBJ
(EXAMPLE.LIS)
libraries
$ LINK EXAMPLE
The LINK command assumes the file type of an input file is OBJ.
(If you use the /MAP qualifier, the linker creates a map file.)
Link the object module
EXAMPLE.EXE
(EXAMPLE.MAP)
$ RUN EXAMPLE
The RUN command assumes the file type of an image is EXE.
Run the executable image
ZK-1016-82

Figure 1-1: Program Development Process

Each command includes a VAX/VMS *file specification* (file-spec) and can also include optional qualifiers. Command qualifiers provide the system with additional information on how to execute the command. See the *VAX-11 PASCAL User's Guide* and the *VAX/VMS Command Language User's Guide* for more information on qualifiers.

The file specification tells the operating system which file to process. A full VAX/VMS 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
```

Often, you need only specify the *file name.* The file name identifies the file and can be up to nine alphanumeric characters long. The *file type* describes the kind of data in the file and can be up to three alphanumeric characters long.

Several files can have the same file name as long as their file types are different. For instance, the file name EXAMPLE is used throughout the following sections to illustrate the four commands involved in developing programs. However, by default, each command applies a different file type to the file name EXAMPLE.

1.1.1 Creating the Program

When you write a program, you must create a VAX/VMS 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 EXAMPLE and a file type of PAS, you can issue the EDIT/EDT command as follows:

```
$ EDIT/EDT EXAMPLE.PAS (RET)
Input file not found
[EOB]
*
```

You must include a file type (usually PAS) with the EDIT/EDT command because it assumes no file type by default. The EDIT/EDT command invokes the VAX/VMS default editor, EDT. The asterisk (*) prompt indicates that EDT is ready to accept input. For information on how to use EDT, see the *EDT Editor Reference Manual.*

1.1.2 Compiling the Program

After you create a VAX–11 PASCAL source file, you compile it. To compile a source file called EXAMPLE.PAS, issue the command:

```
$ PASCAL EXAMPLE (RET)
```

You can omit the file type PAS because the PASCAL command assumes that file type as the default.

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 OBJ
- Uses its own defaults when it creates output files (qualifiers on the PASCAL command can override these defaults)

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

```
$
```

If the program does contain errors, however, the PASCAL compiler displays error messages on your terminal. You can use a text editor to correct the errors in your source program.

It is possible for a program to be successfully compiled and, at the same time, to generate warning-level or information-level messages. In that case, the compiler displays the diagnostic messages on your terminal.

For example, there are many information-level diagnostic messages that point out the use of *nonstandard* PASCAL features (that is, VAX–11 PASCAL extensions). These information-level errors do not affect the compilation of a program in any way; they are reported only to flag the use of VAX–11

PASCAL extensions. You can suppress the diagnostic messages for non-standard PASCAL features by using the /NOSTANDARD qualifier with the PASCAL command as follows:

```
$ PASCAL/NOSTANDARD EXAMPLE RET
```

At some installations of PASCAL, the /STANDARD qualifier is enabled by default. If that is the case at your installation, you may wish to suppress such messages with the /NOSTANDARD qualifier.

The /LIST qualifier on the PASCAL command requests the compiler to create a program listing. A program listing includes the program's source statements, line numbers, any error messages that are reported, and other information related to the compilation. For instance, to create a program listing of EXAMPLE, issue the command:

```
$ PASCAL/LIST EXAMPLE RET
```

This command causes the compiler to create a file called EXAMPLE.LIS in addition to the object module EXAMPLE.OBJ. The /LIST qualifier does not direct EXAMPLE.LIS to the line printer. To obtain a printed copy of the program listing, you must use the PRINT command as follows:

```
$ PRINT EXAMPLE RET
```

The PRINT command assumes the default file type LIS.

1.1.3 Linking the Object Module

An object module (for instance, EXAMPLE.OBJ) is not executable. To generate a file that can be executed by the system, invoke the VAX-11 Linker with the LINK command as follows:

```
$ LINK EXAMPLE RET
```

You can omit the file type because the LINK command assumes the file type OBJ by default. The LINK command in this example creates a file named EXAMPLE.EXE, which is an executable image, that is, a file that contains your program in an executable format. The *linker* automatically includes in the executable image any library routines that the compiler has requested for input and output, error handling, and arithmetic function calculation.

1.1.4 Executing the Program

To execute the program EXAMPLE, use the RUN command. When you issue the RUN command, you need to provide only the name of an executable image; the RUN command assumes the file type EXE by default. Thus, to run the program EXAMPLE, issue the RUN command as follows:

```
$ RUN EXAMPLE RET
```

The first time you run a program, it may not execute properly; if it has a bug, or programming error, you may be able to determine the cause of the error by examining the listing file or the output from the program. When you have determined the cause of the error, you can edit your source program and then repeat the compiling, linking, and running steps to test the result.

1.2 A PASCAL Program Example

This section presents an example of a PASCAL program. This program, titled Grocery_Bill, is illustrated in Figure 1–2. Section 1.3 uses this example to illustrate some fundamental PASCAL concepts. The circled numbers in Figure 1–2 are keyed to more detailed explanations in Section 1.3.

You can run the sample program by following the steps outlined in Section 1.1. The VAX/VMS source file you create need not have the same name as the PASCAL program. For instance, you can create a VAX/VMS file called GROC.PAS to contain the program. The program name Grocery_Bill is an identifier known only within the PASCAL program.

You can use the following commands to create, compile, link, and execute the program Grocery_Bill:

```
$ EDIT GROC.PAS
$ PASCAL/NOSTANDARD GROC
$ LINK GROC
$ RUN GROC
```

If you do not include the /NOSTANDARD qualifier on the PASCAL command and /STANDARD is the default qualifier on your installation, the compiler will report information-level messages for each nonstandard feature used in the program. For example, the use of underscore (_) characters in the program Grocery_Bill is a nonstandard feature.

The program Grocery_Bill is an interactive program in that it prompts you for the data it needs, performs calculations, and then prints the results. Specifically, it performs the following steps:

- Prints instructions for entering prices of grocery items
- Reads each price and sums the prices to obtain a subtotal
- Prompts for a yes or no answer to the question "Do you have any coupons?"
- Reads each coupon value that is entered and sums the values
- Subtracts the value of the coupons from the subtotal to obtain a total
- Prints the total

1.3 The Structure of a PASCAL Program

A PASCAL program consists of a *heading* and a *block*. The heading specifies the name of the program and the names of any *external files* the program uses for input and output. The block is divided into two parts: the *declaration section*, which contains data declarations, and the *executable section*, which contains executable *statements*. Figure 1–2 labels each of these parts.

The following sections describe the heading, declaration section, and executable section of the sample program Grocery_Bill. To help clarify Figure 1–2 and the descriptions below, the following list presents some key syntax rules that apply to all PASCAL programs.

```
                    PROGRAM Grocery_Bill (INPUT, OUTPUT); } Program Heading

                   (* Declarations *)
                   TYPE
                      Yes_No = (Yes, No);❶              (* Defines data type Yes_No
                                                           with values Yes and No *)
Declaration
  Section          VAR
                      Item_Price, Total,❷
                      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 *)

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

Figure 1-2: Sample Program Grocery_Bill

Syntax Rules

1. The semicolon (;) and the period (.) are *delimiters* in PASCAL. The semicolon separates successive PASCAL statements. It also terminates the program heading and the 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. The examples in this primer, however, include a semicolon before the word END. This practice makes it easier to add new statements to the end of the program at a later date. The period marks the end of a PASCAL program.
2. The *reserved words* BEGIN and END are also delimiters; they are not statements. BEGIN and END are used to separate the functional parts of a PASCAL program. They specify the beginning and end of the executable section. They also delimit a compound statement. Every BEGIN must be associated with an END.[1] Therefore, make sure that you have a matching END for every BEGIN in your program.
3. PASCAL allows free formatting of program text. You can place statements anywhere on a line, divide a statement across more than one line, and place several statements on one line. However, you cannot divide a name or number between lines or with a space.
4. *Comments* can appear anywhere in a program. Comments are enclosed in braces ({ }). Alternatively, a comment can start with a left parenthesis and asterisk and end with an asterisk and a right parenthesis. Two examples of comments are:

   { What's it all mean? }

   (* This is the alternative form of a comment. *)

   The PASCAL compiler ignores the text between the comment indicators.

The circled numbers in Figure 1–2 are keyed to the circled numbers appearing in Sections 1.3.2 and 1.3.3.

1.3.1 The Program Heading

A PASCAL program always begins with a *program heading*. The heading consists of:

- The reserved word PROGRAM
- The program's name
- The names of any input and output files to be used
- The semicolon delimiter

The heading in the example in Figure 1–2 is:

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

1. However, there are two cases in which an END does not have to be associated with a BEGIN: the CASE statement (see Section 7.3) and the RECORD declaration (see Section 6.2).

The name of the program is Grocery_Bill and it uses the files INPUT and OUTPUT. INPUT and OUTPUT are names known to PASCAL. They specify *text files* that have been *predeclared* (that is, declared in PASCAL). When you run an interactive program, these names indicate that the program uses your terminal for input and output.

1.3.2 The Declaration Section

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

The declaration section can contain the five kinds of declarations listed below. You need not include all of them in a program. The declarations you do include may appear in any order, and a particular kind may appear more than once. However, you may not declare the same *label*, *constant*, *type*, *variable*, *procedure*, or *function* more than once in a block.

- LABEL
- CONST
- TYPE
- VAR
- PROCEDURE and FUNCTION

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

```
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 ❷:

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

This VAR section declares five real variables: Item_Price, Subtotal, Total, Coupon_Amount, 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.

Within the VAR section, you may specify an initial value for any variable. In this program, the variables Subtotal and Coupons are initialized to 0.0 when they are declared. They will each have the value 0.0 when the program begins executing.

1.3.3 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 executable section of Grocery_Bill is shown in Figure 1-3.

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

Figure 1-3: Executable Section of Grocery_Bill

Between BEGIN and END are calls to procedures that read and write data and statements that change the value of variables and control execution.

Several *input* and *output procedures* are used in the program Grocery_Bill. For example, the first two WRITELN procedures ❸ print on the terminal instructions for entering a list of prices. The text within the apostrophes is printed. The third WRITELN procedure ❹ prints text followed by the value of a variable. Again, the text within the 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 ❺ print either the text that is specified in apostrophes, or text and the value of a variable.

The program Grocery_Bill shows three examples of input procedures ❻. 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* ❼. An assignment statement contains three parts — a variable, the assignment operator (:=), and an *expression*:

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 values 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 ❽ is a *conditional statement*. If the expression (Ans = Yes) is true, the statement following the reserved word THEN is executed:

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

The statement following THEN is a *compound statement*. 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 ❾:

```
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, terminating when the value of the variable Item_Price equals 0.0. The semicolon before UNTIL is optional because the UNTIL clause is not another statement; it is part of the REPEAT statement. The second example of a REPEAT statement is ❿:

```
REPEAT
  .
  .
  .
UNTIL EOF (INPUT);
```

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

Figure 1–4 shows a sample run of the program Grocery_Bill.

```
$ RUN GROC
Enter cost of each grocery item. One item per line.
Enter the value 0.0 to terminate list of items.
1.29
2.50
3.49
0.79
2.29
1.20
0.15
1.89
2.19
0.75
1.50
0.0
Subtotal equals -- $ 18.04
Do you have any coupons? Type yes or no and press <RET>. yes RET
Type value of each coupon. One per line.
Type <CTRL/Z> after entering all coupons.
0.15
0.25
0.29
0.70
0.75
^Z
Pay this amount -- $ 15.90
$
```

Figure 1–4: Sample Run of Grocery_Bill

The program Grocery_Bill illustrates a small subset of the VAX–11 PASCAL language. It was designed to give you a feel for how the parts of a PASCAL program fit together. The following chapters describe the VAX–11 PASCAL language in more detail.

Chapter 2
Data Concepts

This chapter presents some PASCAL data concepts. Section 2.1 introduces the PASCAL concept of types. Section 2.2 explains the PASCAL predefined scalar types. Finally, Sections 2.3 and 2.4 introduce the ways variables and expressions are used in a PASCAL program.

2.1 Types

A *type* is a set of values that share certain characteristics. Associated with each type is a set of operations that can be performed on those values. For example, the integers within a particular range constitute a type and the addition operator (+) can be applied to values of that type.

PASCAL associates a type with each of the following entities:

- Constants
- Variables
- Functions
- Expressions

A *constant* is a literal representing a value of a type. For example, the number 4 is a constant in the type consisting of integers. A *variable* is an entity that can assume different values during program execution. A variable's type is the set of values the variable can assume. A *function* is a computation that is associated with a name and that returns a value. The computation is performed when the function is called by a *function designator*. A function's type is the same as that of the values it can return. An *expression* is a constant, a variable, a function, or a combination of these items separated by *operators*. Every expression is associated with a type.

PASCAL types are divided into three categories. These categories are:

- Scalar types
- Structured types
- Pointer types

A value of a *scalar type* is an indivisible unit of data, for example, the integer

4. You use a scalar value as a single unit; that is, there are no parts that can be accessed individually. Scalar types serve as building blocks for structured types.

A *structured type* is a collection of related data components. You can access and manipulate these components individually. VAX-11 PASCAL's predefined structured types include arrays, records, varying character strings, sets, and files. Chapter 6 presents two structured types: the array and the record. The other structured types are described in the *VAX-11 PASCAL Language Reference Manual.*

A *pointer type* allows you to refer to dynamic variables. See the *VAX-11 PASCAL Language Reference Manual* for information on pointer types and dynamic variables.

2.2 Scalar Types

VAX-11 PASCAL defines the following scalar types:

- INTEGER
- REAL
- SINGLE
- DOUBLE
- QUADRUPLE
- BOOLEAN
- CHAR
- UNSIGNED

The *INTEGER* and *REAL types* are used for manipulating numeric data. VAX-11 PASCAL also provides the types *SINGLE, DOUBLE*, and *QUADRUPLE,* to allow you to distinguish between values that are *single-*, *double-*, and *quadruple-precision* real numbers, respectively. The SINGLE type is identical to the REAL type. (Throughout this primer, the term "real type" refers to the REAL, SINGLE, DOUBLE, and QUADRUPLE types collectively, unless otherwise noted.) The *BOOLEAN type* consists of the truth values: FALSE and TRUE. The *CHAR type* is used for manipulating single character data. For example, 'A' is a value of type CHAR. The sections that follow describe the constants in each of these predefined types.

VAX-11 PASCAL also includes the *UNSIGNED type.* Values of the UNSIGNED type consist of an extended set of the nonnegative integers. The UNSIGNED type is fully described in the *VAX-11 PASCAL Language Reference Manual.*

PASCAL allows you to define your own scalar types. For example, the sample program Grocery_Bill in Chapter 1 defines the type Yes_No. The type Yes_No has two constant values, Yes and No. Details on user-defined types are presented in Section 3.5.

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 the values of a scalar type. For example, among the integers, 2 is greater than 1 but less than 3.

The *predefined* scalar types fall into two groups: real types, which were listed above, and the *ordinal types*. The ordinal types are INTEGER, UNSIGNED, CHAR, BOOLEAN, enumerated types (see Section 3.5.1), and subranges of ordinal types (see Section 3.5.2). Each value of an ordinal type corresponds to a unique integer or *ordinal value* that indicates its place in an ordered list of values of that type. PASCAL provides the following function that returns this ordinal value:

```
ORD (x)
```

If x is a constant of an ordinal type, ORD (x) returns the integer representing its ordinal value.

2.2.1 The Type INTEGER

The *INTEGER type* consists of the whole number values ranging from –2,147,483,647 through 2,147,483,647. 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 (+) may precede a positive integer, but is not required.

Some examples of valid PASCAL integer constants are:

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

VAX–11 PASCAL also accepts integer constants in binary, octal, or hexadecimal notation. To use such notation, refer to the *VAX–11 PASCAL Language Reference Manual* for an explanation of the required syntax.

2.2.2 The Type REAL

Numbers of the REAL type include the positive values from 0.29*(10**(-38)) through 1.7*(10**38), the negative values from –1.7*(10**38) through -0.29*(10**(-38)), and the value 0.0. You can express real constants in two ways:

- Decimal notation
- Floating-point notation

2.2.2.1 Decimal Notation — In *decimal notation*, a real constant consists of a minus sign (–) if the number is negative, an integer part, a decimal point, and a fractional part. A plus sign (+) may 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
```

Note that a zero must precede the decimal point of a fractional quantity and must follow the decimal point of a whole number quantity.

2.2.2.2 Floating-Point Notation — *Floating-point notation* is the representation of a real number in the integer.fraction format, followed by a negative or positive exponent. You can use floating-point notation to represent very large or very small real numbers conveniently. 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. Note that 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:

```
43000E-2
0.043E-4
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).

The DOUBLE and QUADRUPLE real data types allow you to represent values with a greater range and/or greater precision. See the *VAX-11 PASCAL Language Reference Manual* for details and examples.

2.2.3 The Type BOOLEAN

The *BOOLEAN type* consists of the truth values FALSE and TRUE. PASCAL orders these values so that FALSE is less than TRUE: ORD (FALSE) equals 0 and ORD (TRUE) equals 1. Two kinds of operators can be used to form Boolean expressions:

- Relational
- Logical

Sections 2.4.2 and 2.4.3 explain how to form Boolean expressions that include *relational* and *logical operators.*

2.2.4 The Type CHAR

You can use the *CHAR type* for manipulating *character* data. A value of type CHAR is a single element of the *ASCII character set.* The ASCII character set consists of upper- 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 that corresponds to each element of the ASCII character set. These values determine how the elements of type CHAR are ordered. For example, the integer 66 corresponds to the uppercase 'B' and the integer 98 corresponds to the lowercase 'b'. Thus, the character 'B' is less than the character 'b'. In the ASCII character set, all uppercase letters have lower ordinal values 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 constant, enclose the value in apostrophes; to specify the apostrophe character, type it twice within apostrophes. Examples of character constants are:

```
'A'
'*'
'3'
'b'
' '   (the space 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 6.1.2.

2.3 Variables

A *variable* is an entity that can assume different values during program execution. In PASCAL, every variable has a name, a type, and a value (once a value is assigned).

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

```
VAR
   Error_Flag : BOOLEAN;
   Item_Price, Total, Subtotal : REAL;
   I, J : INTEGER;
```

This variable section, introduced by the reserved word VAR, declares Error_Flag to be a variable of type BOOLEAN; Item_Price, Subtotal, and Total to be variables of type REAL; and I and J to be variables of type INTEGER.

A variable does not assume a value until the program explicitly assigns it one. One way to assign a value to a variable is with an assignment statement:

```
Total := Subtotal;
```

If the value of Subtotal is defined, this statement will assign the value of Subtotal to the variable Total. The value of Total will then also be defined.

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

2.4 Expressions

An *expression* is a symbol or a group of symbols that PASCAL can evaluate. These symbols can be individual constants, variables, or functions. For example:

```
Item_Price
```

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

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

```
Subtotal + Item_Price
```

This expression is equal to the sum of the values of Subtotal and Item_Price.

PASCAL includes the following types of operators for forming expressions:

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

Every expression has a type. *Arithmetic operators* are used in arithmetic expressions whose values are integers or real numbers. *Relational* and *logical operators* are used in expressions that yield Boolean results.

2.4.1 Arithmetic Expressions

An *arithmetic expression* evaluates to an integer or real value.[1] It can be an integer or real constant, a variable, or a function designator. Alternatively, it can be a series of integer or real constants, variables, and function designators combined with one or more arithmetic operators (shown in Table 2–1). For example, the following expression consists of two variable names and the subtraction operator (–):

```
Subtotal - Coupons
```

This expression equals the value of Subtotal minus the value of Coupons.

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 any fractional part of the result
REM	A REM B	Produce the remainder after dividing A by B
MOD	A MOD B	Produce the modulus of A with regard to B

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 an 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 always produces a real result.

The DIV, REM, and MOD operators can be used only with integer values and always produce integer results. DIV divides one integer by the other and truncates any fraction from the result. REM returns the remainder after dividing one *operand* by the other. MOD computes the *modulus* of the first operand with regard to the second.

The result of the operation I MOD J is defined only when J is a positive integer. This result is always an integer from 0 through J–1. I MOD J is computed as follows:

- If I is greater than J, J is subtracted repeatedly from I until the result is a positive integer less than J.

1. In this section, the term "real" refers to the REAL and SINGLE types. The rules for using values of types DOUBLE and QUADRUPLE in arithmetic expressions are slightly different from those for types REAL and SINGLE. See the *VAX-11 PASCAL Language Reference Manual* for information on using values of the types DOUBLE and QUADRUPLE in expressions.

- If I is less than 0, J is added repeatedly to I until the result is a positive integer less than J.
- If I is less than J or equal to 0, the result of I MOD J is I.

For example, 5 MOD 3 = 2, (-4) MOD 3 = 2, and 2 MOD 5 = 2.

When both operands are positive, the REM and MOD operators return the same result. For example, 28 REM 5 = 3 and 28 MOD 5 = 3. When the first operand is negative, REM produces a nonpositive result, while MOD produces a nonnegative result. For example, (-42) REM 8 = -2 and (-42) MOD 8 = 6.

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,	R op I	R	4.2 ** 2	1.764E+01
multiplication, exponentiation[2])	I op R	R	4 * 4.5	1.800E+01
	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, REM, MOD	I op I[3]	I	42 DIV 5	8
(division with			4 DIV 5	0
truncation, remainder,			32 REM 5	2
modulo class)			(-4) REM 3	-1
			32 MOD 5	2
			(-4) MOD 3	2

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 an negative integer, you can get unexpected results. Refer to the *VAX-11 PASCAL Language Reference Manual* for the rules governing PASCAL's evaluation of expressions containing negative integer exponents.

3. In the MOD operation, the second operand must be a positive integer.

You can combine operators to form complicated expressions. For example, if all of its operands are integers, 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)
```

The order in which operands are combined is determined by PASCAL's *precedence rules*, described in Section 2.4.4.

2.4.2 Relational Expressions

A *relational expression* tests a specified relationship between two values. It returns TRUE if the relationship is true 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, with the exception that real numbers and integers can be compared. The relational operators can also be used with expressions of types UNSIGNED and SET. See the *VAX-11 PASCAL Language Reference Manual* for details.

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

Note that in the 2-character operators (<>, >=, and <=), the characters 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 Chapters 4 and 7). For example, the program Grocery_Bill contains the following statement:

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

The statements within BEGIN and END are executed only if the expression (Ans = Yes) 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 holds the value 72, the value of the expression will be 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.

2.4.3 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 will be 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 used as 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 value of the integer variable I is odd and FALSE if the value of I is even. Both operands, Flag and ODD (I), must be TRUE for the expression to be TRUE.

Another example:

```
(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 will be TRUE.

2.4.4 Precedence Rules for Operators

When evaluating expressions that contain more than one operator, PASCAL follows *precedence rules* to determine the order in which operands are to be combined. An operation with higher precedence is evaluated before an operation with lower precedence. For example, in the following expression, some operations are performed before others:

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

The division and multiplication operations are performed before the addition. For example, if A equals 4 and B equals 2, A/B will be evaluated to return 2.0; 3 will be multiplied by 4 to return 12. Then, the results of these calculations will be added together to produce 14.0.

Table 2–5 lists the order of precedence of arithmetic, relational, and logical operators, from highest 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. Operations enclosed in parentheses are combined first, regardless of the precedence of operators.
2. Two operators of equal precedence (such as DIV and *) are combined from left to right.

For example, 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 it squares the result (which is 12) to obtain 144; finally, it performs the DIV operation 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
```

Unless X and B are of type BOOLEAN, an error occurs because AND applies only to Boolean operands. For correct evaluation, 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
```

This expression causes PASCAL to generate an error because 4 is not a Boolean value.

Parentheses also help to clarify an expression. A long expression is easier to read if it contains parentheses that indicate 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

You must declare or define every data item you use in a PASCAL program. All declarations and definitions must appear in the declaration section. The declaration section can contain the following parts or sections:

- LABEL — declares *labels* for use by the GOTO statement
- CONST — defines symbolic *constants*
- TYPE — creates *user-defined type* names
- VAR — declares *variables* and their types
- PROCEDURE and FUNCTION — declare *procedures* and *functions* (collectively called *routines*)

A program need not include all these sections. Those sections that are present may appear in any order and can occur more than once per declaration section. Thus, you can use the names LABEL, CONST, TYPE, VAR, PROCEDURE, and FUNCTION as many times per block as you wish. However, you can define or declare the same item only once in a declaration section.

All of these sections except the LABEL section introduce *symbolic names* that represent data items. Section 3.1 describes symbolic names, including identifiers for variables, constants, and so forth. Sections 3.2 through 3.4 explain three parts of the declaration section — CONST, TYPE, and VAR. Section 3.5 shows how to create user-defined scalar types. PROCEDURE and FUNCTION sections are discussed in Chapter 8.

The LABEL section is described in the *VAX-11 PASCAL Language Reference Manual.*

3.1 Symbolic Names

Symbolic names are the words used in a PASCAL program. Symbolic names can be defined by PASCAL or they can be created by the user. For example, the following line of a PASCAL program contains three symbolic names:

```
VAR
   Ans : Yes_No;
```

The word 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:

- Reserved words
- Predeclared identifiers
- User identifiers

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

3.1.1 Reserved Words and Predeclared Identifiers

Reserved words and *predeclared identifiers* are defined by PASCAL and have a special meaning to the compiler. They are printed in uppercase letters in this primer; however, PASCAL does not distinguish between upper- and lowercase letters in reserved words and predeclared identifiers.

3.1.1.1 Reserved Words — PASCAL sets aside certain reserved words that cannot be redefined. Some of the reserved words already shown in this primer are:

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

Appendix A contains a complete list of the VAX–11 PASCAL reserved words.

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

If you choose to redefine one of these identifiers, you should do so with caution. Once a predeclared identifier is used to denote some other item, it can no longer be used for its original purpose within the same block. You could, for example, create a variable named COS; then, however, you could no longer use the predeclared cosine function, COS, which is a useful language feature.

Some of the predeclared identifiers that have been mentioned so far in this text are listed below:

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

Appendix A presents a complete list of VAX-11 PASCAL predeclared identifiers.

3.1.2 User Identifiers

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

When forming an identifier, you must follow VAX-11 PASCAL's syntax rules. An identifier can be a combination of upper- and lowercase letters, digits, dollar sign ($) characters, and underscore (_) characters, with the following restrictions:

- An identifier cannot start with a digit.
- The first 31 characters of every identifier must be unique.
- An identifier cannot contain any blanks.
- Upper- and lowercase letters are considered equivalent.

Although identifiers can be of any length, you will get a warning message at compile time if an identifier exceeds 31 characters. PASCAL recognizes only the first 31 characters; therefore, two 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 suggest the role that the data item is to play. Such a practice enhances the readability of your program. For example, although the word Slug is a valid identifier, it would not be very descriptive as the name of a variable that holds the result of a square root calculation. A variable name like Square_Root, on the other hand, indicates what data that variable holds.

Some examples of valid user identifiers in VAX-11 PASCAL are:

```
Subtotal
Item_Price
Math_Scores
Fica_Tax
```

Examples of invalid user 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 are treated as the same identifier because their first 31 characters are identical.

```
Spring_Inventory_Identification_Tags
Spring_Inventory_Identification_Number
```

Although VAX-11 PASCAL allows the dollar sign ($) character in identifiers, this character has a special meaning to the VAX/VMS operating system in some contexts. For example, all system services and run-time library procedures include a dollar sign in their names. Therefore, you should restrict the use of the dollar sign to identifiers intended to refer to VAX/VMS names.

3.2 Constant Definitions

You can define identifiers to represent constant values in a CONST part of the declaration section. Identifiers and their corresponding values are called *symbolic constants.* The corresponding values can be represented by expressions, which must be *constant expressions.* For instance, a program that adds apples to oranges might use the number 100 to indicate the maximum number of fruits that can be summed. Instead of using the number 100 throughout the program, you can define an identifier and assign it the value 100 as follows:

```
CONST
   Max_Fruits = 100;
```

The identifier Max_Fruits is more descriptive of the constant's use in the program than is the number 100.

Suppose that, instead of giving Max_Fruits a constant value of 100, you want to give it a constant value equal to the sum of the maximum number of apples and the maximum number of oranges. If you know that Max_Apples is equal to 60 and Max_Oranges is equal to 40, you could define the following symbolic constants:

```
CONST
   Max_Apples = 60;
   Max_Oranges = 40;
   Max_Fruits = Max_Apples + Max_Oranges;
```

Max_Apples and Max_Oranges must be declared as symbolic constants before they can be used in a constant expression.

You can define any number of symbolic constants in the CONST sections of your program. The format of the CONST definition is:

```
CONST
     {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 6.1.2), a Boolean constant, or another symbolic constant. The value can also be an expression composed of symbolic constants previously defined in the program. You must separate successive constant definitions with semicolons.

The type of a symbolic constant is the type of its corresponding value. Thus, the type of Max_Fruits shown above is 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. However, remember that the identifier represents a constant value or expression that cannot be changed with subsequent assignment statements or input procedures.

For example, to define a symbolic constant representing the number of students in a class (say, 25), you could 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 would simply modify the CONST definition as follows:

```
CONST
   Class_Size = 28;
```

Changing the CONST definition is easier than changing every occurrence of the value in the program.

More examples of constant definitions are:

```
CONST
   Leap_Year = TRUE;
   Year = 1984;
   Century = 20;
   Dot = '.';
   Country = 'United States';
   Citizenship = Country;
   Num_States = 50;
   Pi = 22.0/7.0;
```

This CONST section defines eight constant identifiers. The identifiers Year, Century, and Num_States represent integers. Leap_Year is equal to the Boolean value TRUE. Dot and Country represent a character value and a character string, respectively. Citizenship is defined to be equal to the symbolic constant Country and thus represents the same character string. Finally, Pi represents the real number resulting from the division of 22.0 by 7.0.

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.

The format is:

```
TYPE
   {type-name = type-definition};...
```

Each type name is a user identifier that indicates the name of the type. The type definition specifies any valid PASCAL type.

This primer covers the following kinds of type definitions:

- Predefined scalar
- Enumerated
- Subrange
- Array
- Record

Enumerated and subrange types are varieties of user-defined scalar types; Section 3.5 explains how to define new scalar types. Chapter 6 shows how to specify the structured array and record types. Refer to the *VAX-11 PASCAL Language Reference Manual* for information on how to specify varying character string, set, file, and pointer types.

3.4 Variable Declarations

Every *variable* in a PASCAL program must be declared before it is used. Section 3.5 shows how to declare variables of user-defined types. Chapter 6 shows how to declare array and record variables.

A variable declaration creates a variable and associates it with an identifier and a type. The identifier and the type are permanent characteristics of the variable. Unless a variable is initialized, its value is undefined until it is assigned a value in the executable section.

You declare variables in the VAR section of a program. For example, the following variable declarations appear in the program Grocery_Bill:

```
VAR
   Item_Price, Total,
   Coupon_Amount : REAL;
   Ans : Yes_No;
```

This VAR section declares three variables of type REAL and one variable (Ans) of type Yes_No. Note that you can declare several variables in the same VAR section.

The format of the VAR section is:

```
VAR
   { {variable-name},... : type };...
```

The variable name can be any valid user identifier. The type can be any of the predefined scalar types — INTEGER, REAL, SINGLE, DOUBLE, QUADRUPLE, BOOLEAN, CHAR, or UNSIGNED. In addition, the type can be any identifier previously defined in the TYPE section or a type definition as outlined in Section 3.3.

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 Boolean variables Error_Flag and Test; the character variable Initial; the real variables Cost and Retail_Pr; and, finally, the integer variables Count, Iterations, I, and J.

VAX-11 PASCAL allows you to assign an initial value to a variable when you declare the variable in a VAR section. For example, the program Grocery_Bill contains the following declarations and *value initializations:*

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

The VAR section assigns the value 0.0 to the variables Subtotal and Coupons. When the program starts executing, those variables assume the value 0.0.

If a variable is not initialized in the VAR section, its value is undefined when the program begins execution. Therefore, the values of Item_Price, Total, and Coupon_Amount are undefined at the beginning of Grocery_Bill. If you tried to use them, the result would be unpredictable.

The format of the VAR section with value initialization is:

```
VAR
   {variable-name : type := value};...
```

The variable name and the type declare a scalar or structured variable. The value is the initial value that is to be assigned to the variable. Note that the operator used for initializing variables is the assignment operator (:=), not the equal sign (=).

The value must be a constant of a type that can be assigned to the variable (including symbolic constants and constant expressions). For example, you can initialize a Boolean variable with either TRUE or FALSE and a character variable with a single character enclosed in apostrophes.

The following example shows a VAR section with value initializations:

```
VAR
   Course : INTEGER := 101;
   Section : CHAR := 'A';
   First_Try : BOOLEAN := TRUE;
   Second_Try : BOOLEAN;
   QPA : REAL := 0.0;
   Temperature : INTEGER := -10;
```

Note that one of the above variables, Second_Try, was not initialized. You do not have to initialize all (or any) of the variables that you declare.

3.5 User-Defined Scalar Types

PASCAL provides the predefined scalar types — INTEGER, REAL, SINGLE, DOUBLE, QUADRUPLE, BOOLEAN, CHAR, and UNSIGNED. In addition, PASCAL allows you to define your own scalar types. For example:

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

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

There are two classes of user-defined scalar types:

- Enumerated
- Subrange

To define an *enumerated type,* you list the type's constant values 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. You may not define a subrange of a real 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 through 100.

You can define a scalar type in either of two parts of the declaration section:

- The TYPE section
- The VAR section

When you define a type 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 name, you can define more than one variable of the type, as follows:

```
VAR
   Ans, Ans1, Ans2 : 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. The subrange 0..100 has no type name in this example.

3.5.1 Enumerated Types

An *enumerated type* is an ordered set of values denoted by constant identifiers. To define an enumerated type, you list the identifiers that represent the constant values of the type. The format of the enumerated type definition is:

({identifier},...)

As with other scalar types, the values in enumerated types are ordered. In particular, they are arranged in ascending order from left to right. For example:

```
TYPE
   Month = (Jan, Feb, Mar, Apr, May, June,
            July, Aug, Sept, Oct, Nov, Dec);
```

By this definition, the relational expression

```
Mar < Oct
```

is TRUE because Mar precedes Oct in the list of values.

The enumerated type definition associates an ordinal value with each value in the type. The ordinal value of the first value listed is 0; the ordinal value of the second value is 1; and so forth. You can apply the ORD function to values of enumerated types. For example, using the type Month from above, the following function is valid:

```
ORD (Aug)
```

This function returns the integer value 7 because Aug is the eighth value listed in the type definition.

A constant identifier (for example, Feb) can be used in only one enumerated type definition. An example demonstrates the reason for this restriction. Suppose the following types were defined in the type section:

```
TYPE
   Month = (Jan, Feb, Mar, Apr, May, June,
            July, Aug, Sept, Oct, Nov, Dec);
   Fiscal_Year = (July, Aug, Sept, Oct, Nov, Dec,
                  Jan, Feb, Mar, Apr, May, June);
```

The second definition is illegal because it would make the following relational expressions ambiguous:

```
Jan < Dec

Feb > July
```

The values of these relational expressions depend on which type is being referenced, Month or Fiscal_Year.

To use the type Month as defined above, you must declare a variable of this new type:

```
VAR
   Birth_Month : Month;
```

The variable Birth_Month can assume any of the values of type Month.

You can define a type in the variable section. For example:

```
VAR
   Ocean :  (Atlantic, Pacific, Indian, Arctic);
```

This declaration creates the variable Ocean, which can take on the values Atlantic, Pacific, Indian, and Arctic. Ocean is a variable name, not a type name.

To initialize a variable of an enumerated type, specify a constant value as you would for a predefined scalar type. For example, you can initialize Ocean as follows:

```
VAR
   Ocean : (Atlantic, Pacific, Indian, Arctic) := Atlantic;
```

The variable Ocean takes on the initial value Atlantic.

Examples

```
TYPE
   Cities = (New_York, Chicago, Los_Angeles, Philadelphia,
             Seattle, Boston, San_Francisco, Washington_DC,
             Dallas, Pittsburgh);
   Colors = (Red, Yellow, Blue, Orange, Purple, Green);

VAR
   Day : (Sun, Mon, Tue, Wed, Thu, Fri, Sat) := Fri;
   Hometown : Cities := Boston;
   Location : Cities;
   Paint, Room_Color : Colors;
```

This TYPE section defines the types Cities and Colors, listing all the values that variables of each type can assume. The VAR section declares the variable Day and defines the values it can assume. The VAR section also declares the variables Hometown and Location of type Cities and the variables Paint and Room_Color of type Colors. The variables Hometown and Day are initialized with the values Boston and Fri, respectively.

3.5.2 Subrange Types

A *subrange type* is a subset of an existing ordinal type called the *base type.* If you know a variable will never use the whole range of values allowed by a base type, you can define a subrange of that base type. For example, a variable that records the number of days per year during which it rains in a certain area can never assume a value that is less than 0 or greater than 366. Thus, you could declare an ordinal subrange type whose values range from 0 to 366:

```
VAR
   Rain_Days : 0..366;
```

The format of the subrange type definition is:

lower-limit..upper-limit

Lower-limit and upper-limit specify the bounds of the subrange; that is, the constant values at the extremes of the subrange interval. The bounds must be constants of the same base type. The base type can be any enumerated or predefined ordinal type. Lower-limit must be less than or equal to upper-limit.

You can use a value of a subrange type anywhere in the program that you can use its base type. Thus, you can use values of the subrange 0..366 in arithmetic expressions just as you can use integers. The ordinal value (returned by the ORD function) of a value of a subrange type is the same as it would be for the base type. For example, in a subrange type consisting of the characters 'A' through 'Z', the ordinal value of 'A' is 65, just as it is in the type CHAR.

You can initialize a variable of a subrange type by specifying a value of the same type in a VAR section of your program. For example, you could initialize the variable Rain_Days as follows:

```
VAR
   Rain_Days : 0..366 := 109;
```

Rain_Days has the value 109 when the program begins executing.

Examples

```
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. 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.

2.
```
TYPE
   Days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
   Colors = (Red, Yellow, Blue, Orange, Purple, Green);
   Primary_Colors = Red..Blue;

VAR
   Week : Days := Wed;
   Spectrum : Colors;
   Paints : Primary_Colors := Green;
   Work_Days : Mon..Fri := 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'.

The value Wed initializes the variable Week, the value Green initializes the variable Paints, and the value Fri initializes the variable Work_Days.

Chapter 4
Fundamental 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, identifiers, operators, expressions, and other statements. This chapter introduces the following statements:

- Assignment statement
- Compound statement
- Control statements
  - IF-THEN statement
  - IF-THEN-ELSE statement
  - FOR statement

The *assignment statement* gives a value to a variable. The *compound statement*, delimited by BEGIN and END, groups other PASCAL statements for sequential execution as a single statement. The IF-THEN, IF-THEN-ELSE, and FOR statements are *control statements*. In the absence of control statements, PASCAL statements are executed in the sequence in which they appear in the source program. Control statements alter this sequence of execution depending on whether specified conditions are met.

As mentioned in Chapter 1, the semicolon (;) is a delimiter used to separate successive PASCAL statements. As such, it is not needed (although PASCAL will accept it) after a statement that is followed by a program element other than a statement — for example, the END delimiter.

4.1 The Assignment Statement

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

```
variable := 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. You can assign 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, SINGLE, DOUBLE, or QUADRUPLE; an expression of type REAL or SINGLE to a variable of type DOUBLE or QUADRUPLE; and an expression of type DOUBLE to a variable of type QUADRUPLE. You can also assign to a subrange variable a value in the specified subrange of its base type. Note that the assignment 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 arithmetic, relational, and logical expressions described in Section 2.4.

For example, suppose you make 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 can therefore 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 the corresponding variable.

4.2 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
{statement};...
END

The statements between the BEGIN and END delimiters can be any PASCAL statements, including other compound statements. Successive statements must be separated with a semicolon; however, no semicolon is required between the last statement and the END delimiter. PASCAL treats the compound statement as if it were a single statement. For example, the program Grocery_Bill contains a compound statement that is part of an IF-THEN statement:

```
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;
```

If the Boolean expression (Ans = Yes) is TRUE, every statement between BEGIN and END will be executed; if the expression is FALSE, the flow of control will transfer to the statement following the END.

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

4.3 The IF-THEN Statement

Often you want a statement to be executed only if a certain condition is satisfied. The IF-THEN statement causes the conditional execution of a statement; that is, it executes a given statement if a specified Boolean expression is TRUE. The format is:

IF Boolean-expression
THEN
    statement

The meaning of this statement is suggested by the English words "if" and "then." "If" the expression is TRUE, "then" the statement will be executed. If the expression is FALSE, program control will pass to the statement following the IF-THEN statement.

Together 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.

The flow of control for the IF-THEN statement is illustrated in Figure 4-1.

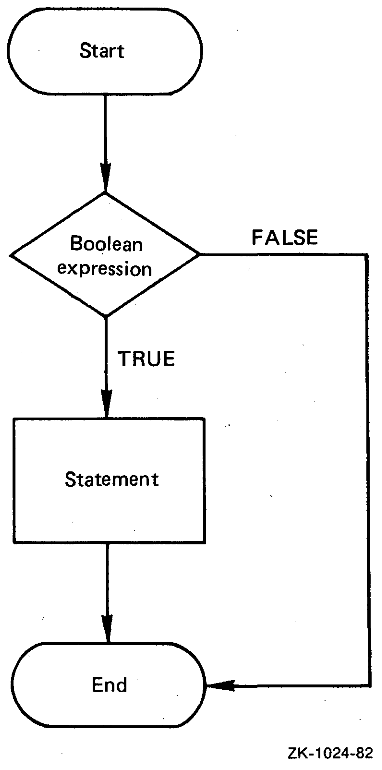

Figure 4-1: The IF-THEN Statement Flow Chart

For example:

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

If the value of A is less than 0, the value of Neg_Ints will be increased by 1.

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, had there been a semicolon after THEN, the assignment statement would have been executed regardless of the value of the Boolean expression.

Examples

1.
```
IF (Ans = Yes)
THEN
   BEGIN
      .
      .
      .
   END;
```

The object of the THEN clause can be a compound statement. The statements between BEGIN and END will be executed if the Boolean expression is TRUE. If the expression is FALSE, execution will continue with the statement following the END.

2.
```
IF (Ch >= 'A') AND (Ch <= 'Z')
THEN
   BEGIN
   Letter := TRUE;
   Letter_Total := Letter_Total + 1;
   END;
```

If both relational expressions are TRUE, the compound statement will be executed.

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 will write the message in parentheses if the current value of Errorflag is TRUE.

4.4 The IF-THEN-ELSE Statement

The IF-THEN-ELSE statement is an alternative form of the IF-THEN statement. The IF-THEN-ELSE statement causes the program to select and execute one of two statements depending on the value of a Boolean expression. 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 will be executed only if the expression is TRUE. If the expression is FALSE, statement2 will be executed instead. Together, the reserved word ELSE and statement2 are called the ELSE clause of the IF-THEN-ELSE statement. The flow of control for the IF-THEN-ELSE statement is shown in Figure 4–2.

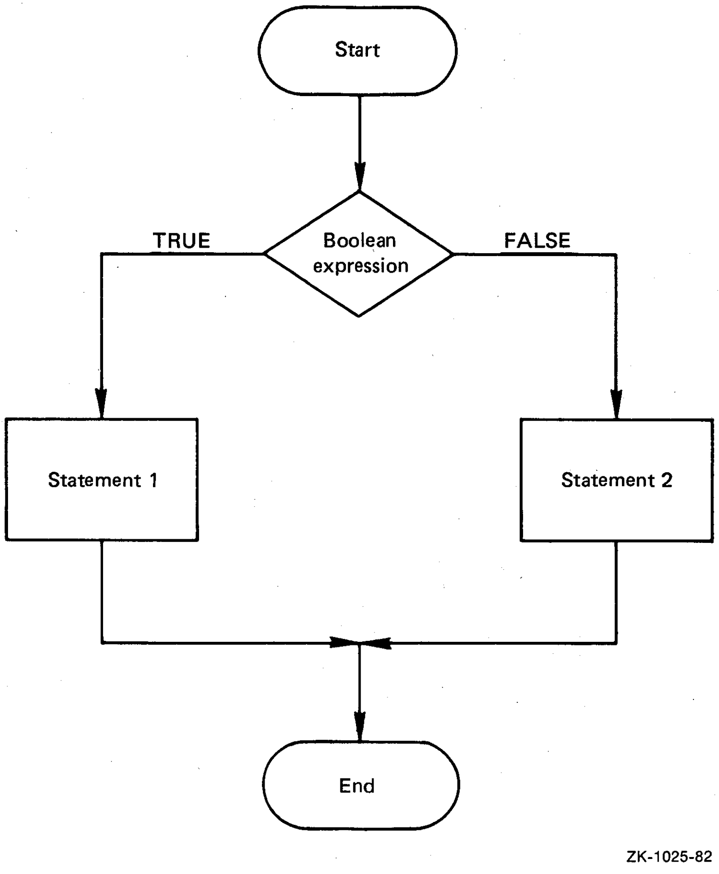

Figure 4–2: The IF-THEN-ELSE Statement Flow Chart

You must not place a semicolon between statement1 and the word ELSE. The IF-THEN-ELSE statement is a single statement and the IF-THEN clause cannot be separated from the ELSE clause. You will 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 wanted the ELSE clause to be associated with the first IF-THEN statement, you would 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 will be executed and the ELSE clause will be ignored. If A is not equal to 1, 1 will be assigned to D regardless of the value of B.

Examples

1.
```
IF (Last_Initial >= 'A') AND (Last_Initial <= '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 names start with A through M, and on the 28th to customers whose last names start with N through Z.

2.
```
IF (Card_Sum > 21)
THEN
   Lose := TRUE
ELSE
   IF (Card_Sum >= 17)
   THEN
      Deal := FALSE
   ELSE
      BEGIN
      Deal := TRUE;
      Card_Sum := Card_Sum + 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 will lose. If the sum is between 17 and 21, inclusive, the player will not be dealt any more cards. If the sum is less than 17, the player will be 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 is to use PASCAL's conditional CASE statement, which is explained in Section 7.3.

4.5 The FOR Statement

The FOR statement controls a *loop*. A loop is a construct containing a statement or a series of statements that is executed repetitively until a certain condition is met. The repetitively executed statement or series of statements is called the *loop body*. In the FOR loop, the loop body is executed repetitively based on the value of an automatically incremented or decremented variable.

For example:

```
FOR Int := 1 TO 10 DO
   BEGIN
   Square := Int * Int;
   WRITELN ('The square of', Int:3, ' equals', Square:4);
   END;
```

The statements between BEGIN and END are executed 10 times. The first time through the loop, the variable INT has the value 1, the second time it has the value 2, and so on. For each integer from 1 to 10, this example computes the integer's square and writes it with a message to the terminal. Thus, the output from the first iteration would be:

```
The square of  1 equals    1
```

The format of the FOR statement is:

FOR control-variable :=initial-value $\left\{\begin{array}{c}\text{TO}\\ \text{DOWNTO}\end{array}\right\}$ final-value DO
statement;

The *control variable* can be a variable of any ordinal type; it cannot be the name of an array component or a record field. (Arrays and records are discussed in Chapter 6.) The initial and final values must be expressions of the same type as the control variable.

The initial and final values are computed only once, at the beginning of the FOR statement. Thus, if the loop body changes the value of the final expression, the change will not affect the number of times the loop is executed.

The loop body is repetitively executed while the control variable ranges from the initial value to the final value. In the TO form of the FOR statement, the final value must be greater than or equal to the initial value to cause the loop body to execute. In the DOWNTO form, the final value must be less than or equal to the initial value. In either form, if the initial and final values are equal, the loop body is executed once.

The loop body must not change the value of the control variable. After the FOR statement is executed, the control variable is left undefined; you cannot assume that it retains a value. Therefore, you must assign a new value to the control variable before you can use it elsewhere in the program.

The flow of control for both forms of the FOR statement is shown in Figure 4-3.

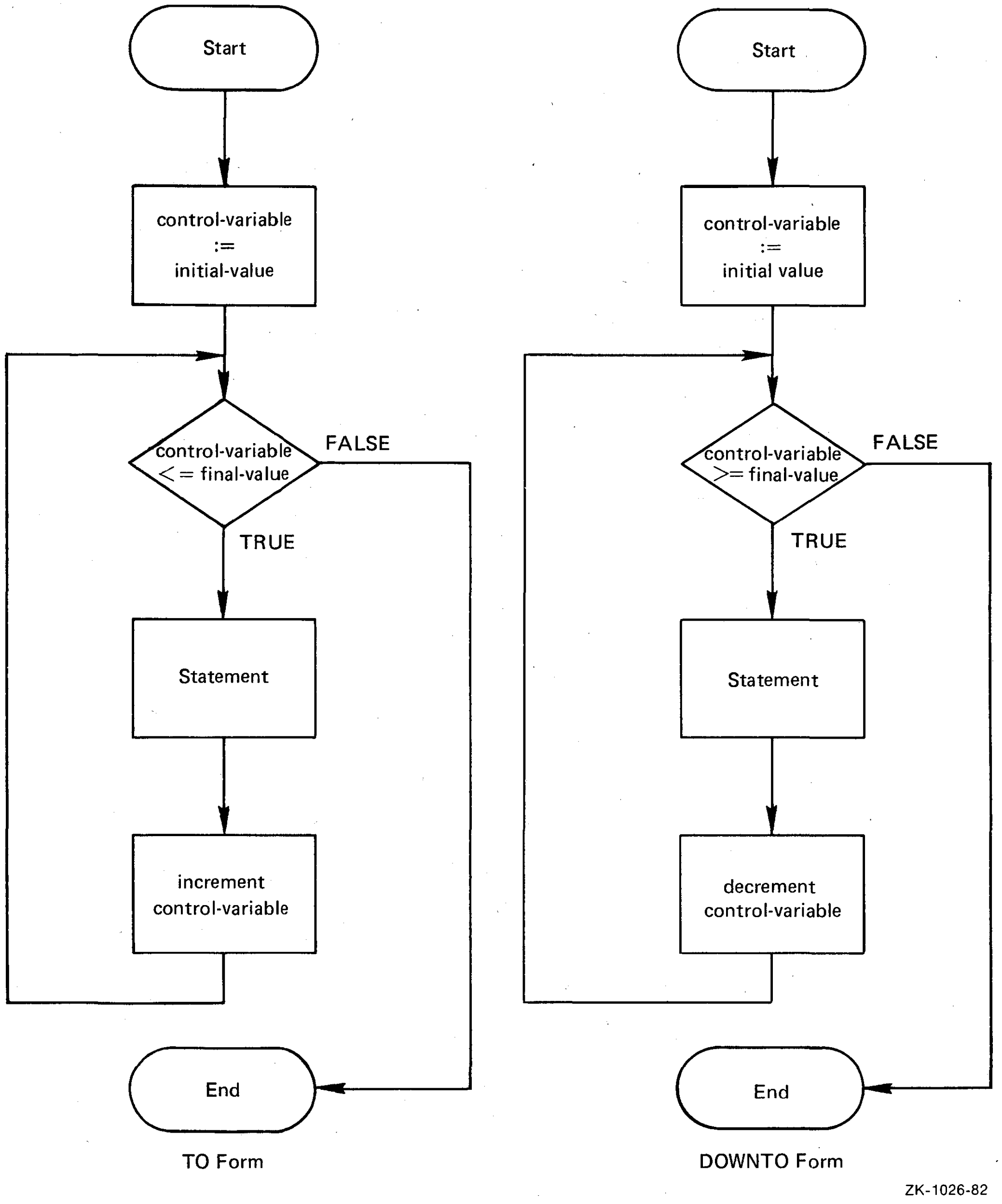
Start
control-variable := initial-value
control-variable <= final-value
FALSE
TRUE
Statement
increment control-variable
End
TO Form
Start
control-variable := initial value
control-variable >= final-value
FALSE
TRUE
Statement
decrement control-variable
End
DOWNTO Form
ZK-1026-82

Figure 4-3: The FOR Statement Flow Charts

In the TO form, on each iteration of the loop, the control variable is assigned the *successor value* in its type. That is, if the control variable is an integer, the FOR statement adds 1 to the control variable's value upon each iteration. For control variables of other types, the control variable takes on a successive value of its type at each iteration. Similarly, in the DOWNTO form, on each iteration the control variable is assigned the *predecessor value* in its type.

Examples

1.
```
TYPE
   Days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);

VAR
   Current_Day :  Days;
   Hours_Worked :  REAL;

FOR Current_Day := Sun TO Sat DO
   IF (Current_Day <> Sun) AND (Current_Day <> Sat)
   THEN
      Hours_Worked := Hours_Worked + 8;
```

This example shows the use of a FOR statement with a control variable of an enumerated type. On each iteration of this loop, the variable Current_Day is assigned the successor of its current value in the type Days. Thus, Current_Day equals Sun on the first iteration, Mon on the second, and so forth. On the last iteration, the value Sat is assigned to Current_Day, the loop body is executed for the final time, and control moves to the statement following the loop.

2.
```
FOR Student_Id := 25 DOWNTO 1 DO
   BEGIN
   READLN (Grade1, Grade2, Grade3);
   Average := (Grade1 + Grade2 + Grade3) DIV 3;
   IF Average >= 65
   THEN
      WRITELN ('Student number ', Student_Id, ' passed')
   ELSE
      WRITELN ('Student number ', Student_Id, ' failed');
   END;
```

This example shows the DOWNTO form of the FOR statement. On successive iterations of the loop, the values from 25 down to 1 are assigned to the variable Student_Id. For each student identification number, three grades are read into the variables Grade1, Grade2, and Grade3. The loop body computes a grade average and writes the appropriate message depending on whether the student passed or failed.

Chapter 5
Reading and Writing Data

The PASCAL statements described in the preceding chapter allow a program to manipulate data and perform certain operations. To enter data into a program, the program must perform input operations. To display the results of its actions, the program must perform output operations. This chapter describes VAX–11 PASCAL terminal input and output (I/O); that is, how to read and write data interactively from a terminal.

Specifically, this chapter covers the following topics:

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

5.1 The Predeclared Text Files Input and Output

You perform I/O operations on *file variables*. A variable of the structured type FILE is a sequence of data items, called *file components*, that have the same type.

PASCAL predeclares two text file variables, named 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.

When you run a program in *interactive mode*, VAX–11 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 as the file OUTPUT for writing data.

Because INPUT and OUTPUT are predeclared, you do not declare them in the declaration section. Instead, you specify them in the heading of a program that uses them. For example, for an interactive program that accepts input data from the terminal and writes output data to the terminal (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 it 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 currently be accessed. You can imagine a file's position as a movable window 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.

The *VAX-11 PASCAL Language Reference 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.

5.2 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, that is, your terminal. The READ procedure reads values from a file and assigns them to 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.

5.2.1 The READ Procedure

The READ procedure reads data from the file variable INPUT and assigns the values that are read to the specified variables. For 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},...)

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

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 scalar type, including an enumerated type. As explained in Section 6.1.2, 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. For example, suppose that the variables in the following READ procedure are of type REAL, INTEGER, and INTEGER, respectively:

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

The following values can be read into the specified variables:

```
98 11 75
```

The variable Temp is assigned the value 98, 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 the tab are values of type CHAR, this rule does not apply when you are typing 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 a result of a read operation, the value of the component in the current file position is assigned to a variable; then the file position advances to the file component following the input value.

Examples

1. **Statements** / **Input**

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

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

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

   If these variables are 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 use any number of spaces to separate input values. The values also can appear on different lines as follows:

   ```
   2

   14          1984
   ```

As above, Month equals 2, Date equals 14, and Year equals 1984. Remember that the relative position of the numbers is important. If you typed 14 before 2 at the terminal, the resulting values of the variables Month and Date would be reversed.

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 in a file. The READ procedure reads a character and assigns it to Char_Var. If the character is not a space (′ ′), the variable Count will be incremented by 1. If the character is a space, the assignment statement (Count := Count + 1;) is skipped.

5.2.2 The READLN Procedure

Another PASCAL input procedure is the READLN procedure. The READLN procedure simply performs a READ and then positions the file at the beginning of the next line. For example:

```
READLN (Item_Price);
```

This READLN procedure reads a value into the variable Item_Price and then positions the file at the beginning of the next line. Thus, any remaining data on the input line is ignored. (For this reason, you were instructed to type one item per line in the program Grocery_Bill in Chapter 1.)

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

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

READLN (‖INPUT,‖{variable},...)

or

READLN ‖(INPUT)‖

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 advances to the first component of the next line.

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

```
READLN;
```

This procedure advances the file to the beginning of the line after the current line without reading any values.

Examples

1. **Statements** **Input**

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

The first READLN procedure reads the values 1 and 2 and assigns them to X and Y, respectively. Then the file position advances 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 4 to A and the value 22 to B.

If the first READLN procedure were instead a READ procedure, only the first line of input would be read. READ (X,Y) would read the values 1 and 2 and assign them to X and Y. The file position would not advance. READLN (A,B) would read the values 3 and 4 from the same input line and assign them to A and B. The file position would then advance to the next line to wait for another call to an input procedure.

2. **Statement** **Input**

Statement	Input

`READLN (X,Y,Z)`	`1 100`
	`1000 1001`

This procedure call assigns 1 to X, 100 to Y, and 1000 to Z. Then the file position advances to the beginning of the line following the values 1000 and 1001. Note that the READLN procedure reads across lines until it finds a value for each specified variable; it moves to the next line only after the 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.

5.3 Writing Data

To display the results of its actions, a program must perform output operations. PASCAL provides the WRITE and WRITELN procedures for data output.

By default, the WRITE and WRITELN procedures write data to the file OUTPUT, which is associated with your terminal. The WRITELN procedure simply performs the WRITE procedure, and then positions the file at the beginning of a new line.

5.3.1 The WRITE Procedure

The WRITE procedure writes data to the file variable OUTPUT. For example:

```
WRITE (Total);
```

This procedure call writes the value of the variable Total on your terminal.

The general format, when using the default file variable OUTPUT, is:

WRITE (‖OUTPUT,‖print-list)

Because OUTPUT is the default, you can either 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 simply specify its identifier. You can print the result of an arithmetic, relational, or logical expression by including 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 (ɓ) indicates that the corresponding WRITE procedure prints a space.

```
Statements                              Output

WRITE (IntVar);                         ɓɓɓɓɓɓɓɓ12
WRITE ((2>3) AND Flag);                 ɓFALSE
WRITE ('IntVar equals ', IntVar);       IntVarɓequalsɓɓɓɓɓɓɓɓ12
```

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 '). The value 12 is printed with a default number of leading blanks.

After a WRITE procedure is executed, the file 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.

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 write parameter in the print list. However, without the field width specification, PASCAL uses the default values listed in Table 5-1.

Table 5-1: Default Values for Field Width

Type of Variable	Number of Characters Printed
Integer	10
Real	16
Double	24
Quadruple	32
Boolean	6
Character	1
Enumerated	Size of identifier + 1 up to 32
String	Length of string

For example, the default field width for a real value is 12 characters. If the value of a real variable called Average is 5.5, it is printed as follows:

Statement	**Output**
`WRITE (Average);`	`b5.50000E+00`

Note that real values are printed in floating-point format by default. The value of Average is written in a field of 12 characters (which includes a leading blank).

To override the default for a particular value, you must specify a field width in the print list. The following is the general form of field-width specifications:

write-parameter : minimum ‖: fraction‖

Minimum and fraction represent nonnegative integer expressions. Minimum indicates the minimum number of characters to be printed. Fraction, which is used only with real values, indicates the number of digits to be printed to the right of the decimal point.

For example, you may prefer to print the real-number value of Average in a 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:

```
ThebaveragebisbS5.5
```

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, a leading blank is optional.

The integer 1 specifies that one digit will appear to the right of the decimal point. Thus, the WRITE procedure above specifies a field width of at least four characters, 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 various kinds of write parameters as follows:
   a. Character strings and nonnumeric scalar values are truncated on the right to the specified field width.
   b. Integers and real numbers in decimal format are printed using the full number of characters needed for the value, thus overriding the field-width specification.
   c. Real, double, and quadruple values in floating-point format are printed in a field of at least eight characters (for example, –1.0E+00).

d. All real values in either decimal or floating-point format are printed with a leading minus sign if they are negative. Nonnegative real numbers printed in decimal format need not include a leading blank.

Examples

1. **Statement**

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

Output

```
Firstb̸numberb̸--b̸b̸b̸b̸b̸b̸b̸b̸1
```

Suppose the value of Number is 1. This WRITE procedure prints the text ('First number -- ') followed by eight spaces and the numeral 1. That is, 1 is said to be right-justified in the field of nine characters.

2. **Statement**

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

Output

```
Thisb̸isb̸ab̸te
```

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

3. **Statement**

```
WRITE (Number:4, ' values averaged to ', Average:3:1);
```

Output

```
b̸b̸b̸5b̸valuesb̸averagedb̸tob̸5.5
```

One WRITE procedure can contain several values and character strings as in this example. If Number equals 5 and Average is 5.5, the output shown will be printed. Three leading blanks are included before the number 5 to fill the print field, which is 4. The value of Average is printed in a field of three characters.

4. **Statement**

```
WRITE(Num1:5:1, ' and', Num2:5:1, ' sum to', (Num1+Num2):6:1);
```

Output

```
b̸71.1b̸andb̸29.9b̸sumb̸tob̸101.0
```

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.

5. **Statements**

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

Output

```
FirstƀcolumnƀheadingƀƀƀƀƀƀƀƀƀƀƀƀƀƀSecondƀcolumnƀheading
```

Remember that after a WRITE procedure is executed, the file 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.

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

```
VAR
   Color : (Blue, Yellow, Black,
            Fire_Engine_Green):= Yellow;
```

The WRITE procedure call

```
WRITE ('My favorite color is', Color:15);
```

produces the following output:

```
Myƀfavoriteƀcolorƀis ƀƀƀƀƀƀƀƀƀYELLOW
```

If, however, the value Fire_Engine_Green is assigned to Color, the following appears:

```
MyƀfavoriteƀcolorƀisƀFIRE_ENGINE_GRE
```

Since the field width specified is not wide enough for all 17 characters in FIRE_ENGINE_GREEN, PASCAL truncates the last 2 characters.

5.3.2 The WRITELN Procedure

Another PASCAL output procedure is the WRITELN procedure. The WRITELN procedure simply performs the WRITE procedure, then positions the file 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 they are specified for the WRITE procedure. Furthermore, the field-width rules described in Section 5.3.1 also apply to the WRITELN procedure.

If several parameters occur 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 omission is useful when you want to start a new line or when you want to write a blank line to the output file.

Examples

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 is positioned at the beginning of a new line following the output. In contrast, if you used WRITE procedures instead of WRITELN procedures in the example above, the output from both of the print lists would appear on one line, as follows:

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

The file is positioned immediately after the value 15.

2. **Statements**

```
WRITELN ('Name:', 'Age:':19, 'Soc. Sec. #:':28);
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 of the items in the print list are printed on one line. Then the file position advances to the beginning of a new line. This example also shows how to print a blank line. Because it has no print list, the second WRITELN procedure prints no characters, but creates a blank line.

5.4 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.

5.4.1 The EOLN Function

Text files are divided into lines. Each line ends with a line-separator mark, which indicates 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 file. For the purposes of this chapter, 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 is positioned at the end of a line or as soon as 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 is positioned on the EOLN mark. In contrast, after a READLN procedure reads the last component on a line, the file is positioned at the beginning of the next line, that is, past the EOLN mark. Thus, after a READLN is performed, the EOLN function is never 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, 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;
```

Assume that 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. (Section 7.2 contains information on the WHILE statement.) 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 body is not executed again.

5.4.2 The EOF Function

Every file ends with an end-of-file mark that you can test for with the EOF function. The EOF function is TRUE when the file is positioned on this end-of-file mark. The EOF mark follows the last EOLN mark in a text 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 has been read, EOF becomes TRUE. At all other times, EOF is FALSE.

The diagram in Figure 5-1 represents the characters and the EOLN and EOF marks in a text file.

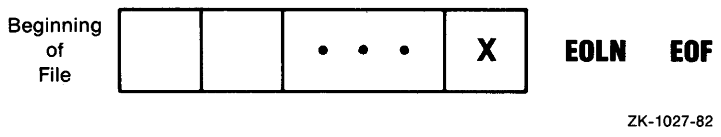

Figure 5-1: The End of a Text File

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. As a result of the READLN procedure call, the file position advances past the EOLN mark. Thus, the file position appears as shown in Figure 5-2.

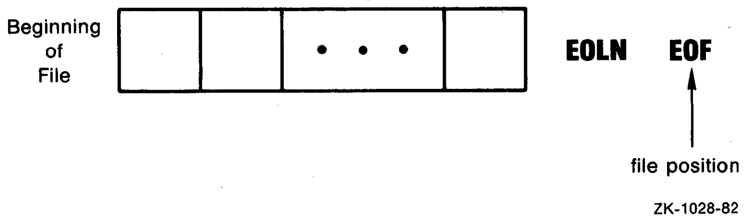

Figure 5-2: File Position at End-of-File

You usually use the READLN procedure before an EOF test. The READLN procedure advances the file position past the EOLN mark (that is, to the EOF mark), as shown in Figure 5-2.

If you use a READ procedure, the last component in the file is read and the file is positioned on the EOLN mark. EOF is not TRUE because the file position has not been advanced to the EOF mark.

The only time EOF can be TRUE after a READ procedure is when a value is being read into a character variable. In that case, after the last component in a file is read, the file is positioned at the EOLN mark. EOF is still FALSE. As a result of one more read operation, a space (' ') is assigned to the character variable and EOF is TRUE.

When you are typing input at the terminal, you can indicate the end of the file by typing a CTRL/Z. CTRL/Z generates an EOLN mark (if the previous component was not 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 to be positioned on the EOF mark.

For example, the program Grocery_Bill contains the following construct:

```
REPEAT
   READLN (Coupon_Amount);
   Coupons := Coupons + Coupon_Amount;
UNTIL EOF (INPUT);
```

In Grocery_Bill, WRITELN procedures directly above this loop print instructions for entering data items and terminating the items entered with a CTRL/Z. Each time the above READLN procedure reads a value for the variable Coupon_Amount, 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, and EOF becomes TRUE. (The REPEAT statement is explained in Section 7.1.)

Chapter 6
Structured Types: the Array and the Record

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. That is, the unit of data contains no smaller data items that can be manipulated individually.

A variable of a *structured type,* on the other hand, is a collection of related data items that you can access and manipulate individually. Although you refer to an entire structured variable with one identifier, you can treat its data items as individual variables.

VAX–11 PASCAL provides the following structured types for building *data structures:*

- Arrays
- Records
- Varying character strings
- Sets
- Files

An *array* is a collection of a specified number of data items of the same type. A *record* is a collection of data items that can be of different types. A *varying character string* is a sequence of ASCII characters whose values are strings of different lengths. A *set* is a collection of data items of an ordinal type. A *file* is a sequence of any number of data items of the same type.

This chapter presents the array (Section 6.1) and the record (Section 6.2) types. A special case of the file type — the *text file* — was introduced in Chapter 5. A detailed presentation of the varying character string, set, and file types, however, is beyond the scope of this primer. Refer to the *VAX–11 PASCAL Language Reference Manual* for more information on these structured types.

6.1 Arrays

An array is a group of data items of the same 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,* enclosed in brackets. The indexes need not be integers; they can be values of any ordinal type.

The following is an example of an array variable 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 indexes. In the array Word, the indexes are a subrange 1..20 of integers.
3. The type of the components. The components of Word are of type CHAR.

The index of an array 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 as variables of the component type. For example, the component Word[1] can appear on the left-hand side of an assignment statement or as a parameter of the ORD function.

The format of the type definition for an array is:

ARRAY[{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 merely the identifier CHAR. However, the index type cannot be the full range of the type INTEGER because an array of that size would occupy too much space in memory.

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 called a *multidimensional array*, as explained in Section 6.1.1.

The type definition shown above can appear in the TYPE section or in the VAR section. An example of defining an array type in the TYPE section is:

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

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

```
VAR
   Sales_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 from 1 to 100. The variable Sales_Items stores the price for each item. Thus, for example, the price for item number 20 is stored in Sales_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);
    Work_Day = 0..24;

VAR
    Work_Week : ARRAY[Mon..Fri] OF Work_Day;
```

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

Suppose you want to write a program to calculate the average of the test scores earned in a particular course. You can treat the group of test scores as an array. The following declarations create an array type and an array variable of that type.

```
TYPE
    Tests = 1..Num_Tests;
    Test_Scores = ARRAY[Tests] OF INTEGER;

VAR
    Score : Test_Scores;
```

Note that you can use a type identifier (for example, Tests) as the index type in an array definition. If Num_Tests is a constant identifier equal to 6, Test_Scores is an array of integers whose index values can range from 1 to 6. To use an individual score in an executable statement, specify the array variable name (Score) and an integer expression whose value is between 1 and Num_Tests.

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

```
PROGRAM Average_Scores (INPUT, OUTPUT);

CONST
   Num_Tests = 6;                (* Number of scores to be averaged *)

TYPE
   Tests = 1..Num_Tests;
   Test_Scores = ARRAY[Tests] OF INTEGER;

VAR
   Score : Test_Scores;
   Sum, I, Average : INTEGER;

BEGIN
Sum := 0;
FOR I := 1 TO Num_Tests 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 Num_Tests;
WRITELN ('The following scores were entered:');
FOR I := 1 TO Num_Tests 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 Num_Tests. 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 following scores were entered:
 100
  88
  75
  90
  63
  94
The average is:  85
```

In an executable statement, you can specify a component of Score with an index that is a variable name. In both FOR loops in the program Average_Scores, 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.

Accessing successive components in an array is a common use of the FOR statement. By merely incrementing the control variable, the FOR statement accesses each component of the array with fewer program statements than if each component were a separate variable. In the program Average_Scores, both FOR loops process each component of Score, that is, Score[1], then Score[2], up through Score[Num_Tests].

The array component Score[I] is used in the same manner as a variable in the READLN procedure, in the expression Sum + Score[I], and in a WRITELN procedure. Score[I] is in fact 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 an array type that represents the temperatures for each day in a month. The executable section in the example shows the assignment of one array to the other.

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

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
0	NUL	Null	40	(	Left parenthesis
1	SOH	Start of heading	41	)	Right parenthesis
2	STX	Start of text	42	*	Asterisk
3	ETX	End of text	43	+	Plus sign
4	EOT	End of transmission	44	,	Comma
5	ENQ	Enquiry	45	–	Minus sign or hyphen
6	ACK	Acknowledgement	46	.	Period or decimal point
7	BEL	Bell	47	/	Slash
8	BS	Backspace	48	0	Zero
9	HT	Horizontal tab	49	1	One
10	LF	Line feed	50	2	Two
11	VT	Vertical tab	51	3	Three
12	FF	Form feed	52	4	Four
13	CR	Carriage return	53	5	Five
14	SO	Shift out	54	6	Six
15	SI	Shift in	55	7	Seven
16	DLE	Data link escape	56	8	Eight
17	DC1	Device control 1	57	9	Nine
18	DC2	Device control 2	58	:	Colon
19	DC3	Device control 3	59	;	Semicolon
20	DC4	Device control 4	60	<	Left angle bracket
21	NAK	Negative acknowledgement	61	=	Equal sign
22	SYN	Synchronous idle	62	>	Right angle bracket
23	ETB	End of transmission block	63	?	Question mark
24	CAN	Cancel	64	@	At sign
25	EM	End of medium	65	A	Uppercase A
26	SUB	Substitute	66	B	Uppercase B
27	ESC	Escape	67	C	Uppercase C
28	FS	File separator	68	D	Uppercase D
29	GS	Group separator	69	E	Uppercase E
30	RS	Record separator	70	F	Uppercase F
31	US	Unit separator	71	G	Uppercase G
32	SP	Space or blank	72	H	Uppercase H
33	!	Exclamation mark	73	I	Uppercase I
34	"	Quotation mark	74	J	Uppercase J
35	#	Number sign	75	K	Uppercase K
36	$	Dollar sign	76	L	Uppercase L
37	%	Percent sign	77	M	Uppercase M
38	&	Ampersand	78	N	Uppercase N
39	'	Apostrophe	79	O	Uppercase O

Table B–1: (Cont.) The ASCII Character Set

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
80	P	Uppercase P	104	h	Lowercase h
81	Q	Uppercase Q	105	i	Lowercase i
82	R	Uppercase R	106	j	Lowercase j
83	S	Uppercase S	107	k	Lowercase k
84	T	Uppercase T	108	l	Lowercase l
85	U	Uppercase U	109	m	Lowercase m
86	V	Uppercase V	110	n	Lowercase n
87	W	Uppercase W	111	o	Lowercase o
88	X	Uppercase X	112	p	Lowercase p
89	Y	Uppercase Y	113	q	Lowercase q
90	Z	Uppercase Z	114	r	Lowercase r
91	[	Left square bracket	115	s	Lowercase s
92	\	Back slash	116	t	Lowercase t
93	]	Right square bracket	117	u	Lowercase u
94	ˆ or ↑	Circumflex or up arrow	118	v	Lowercase v
95	← or _	Back arrow or underscore	119	w	Lowercase w
96	`	Grave accent	120	x	Lowercase x
97	a	Lowercase a	121	y	Lowercase y
98	b	Lowercase b	122	z	Lowercase z
99	c	Lowercase c	123	{	Left brace
100	d	Lowercase d	124	|	Vertical line
101	e	Lowercase e	125	}	Right brace
102	f	Lowercase f	126	~	Tilde
103	g	Lowercase g	127	DEL	Delete

```
(* Declarations *)

CONST
   Days = 31;                   (* Number of days in January *)

TYPE
   Temp = -20..60;              (* Range of temperatures occurring in January *)
   Month_Temp = ARRAY[1..Days] OF Temp;
   (* Month_Temp has 31 components, each is the temp on one day in January *)

VAR
   Sum, I, Average_Temp,        (* Average temp in current January *)
   Record_Ave_Temp : INTEGER;   (* Average temp in January with record lows *)
   Current_Jan, Record_Jan : Month_Temp;
   (* Current_Jan, and Record_Jan represent each day's temperature in this
      year's January and the January 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;
```

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.

6.1.1 Multidimensional Arrays

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
   Class_Size = 15;
   Num_Tests = 5;

TYPE
   Class = 1..Class_Size;
   Tests = 1..Num_Tests;

VAR
   Class_Scores : 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, the variable declaration creates a two-dimensional array called Class_Scores, which 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 index values, one for each dimension. The first index corresponds to the first dimension declared and the second index corresponds to the second dimension declared. 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 6-1.

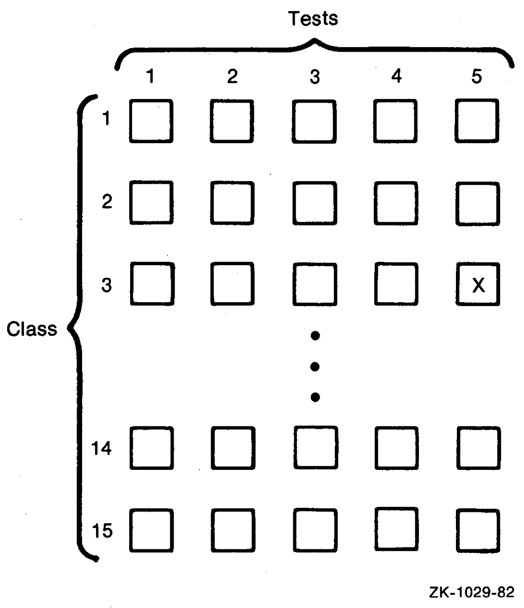

Figure 6-1: The Two-Dimensional Array Class_Scores

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 index 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 6-1 by an X, is found 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. For example, suppose you want to find the average test score for each student in Class_Scores. To process one student's score, the row index should be held constant while the values in each column of that row are averaged. Then, for each successive student, the row index should be incremented and the same averaging operation should be performed on the new row.

The following declaration creates a variable that will hold 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 Class_Size DO
   BEGIN
   Sum := 0;
   FOR J := 1 TO Num_Tests DO
      Sum := Sum + Class_Scores[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 the result of 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 1 to 8. 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.

The floors 1 through 8 in Hotel_Vacancies are illustrated in Figure 6–2.

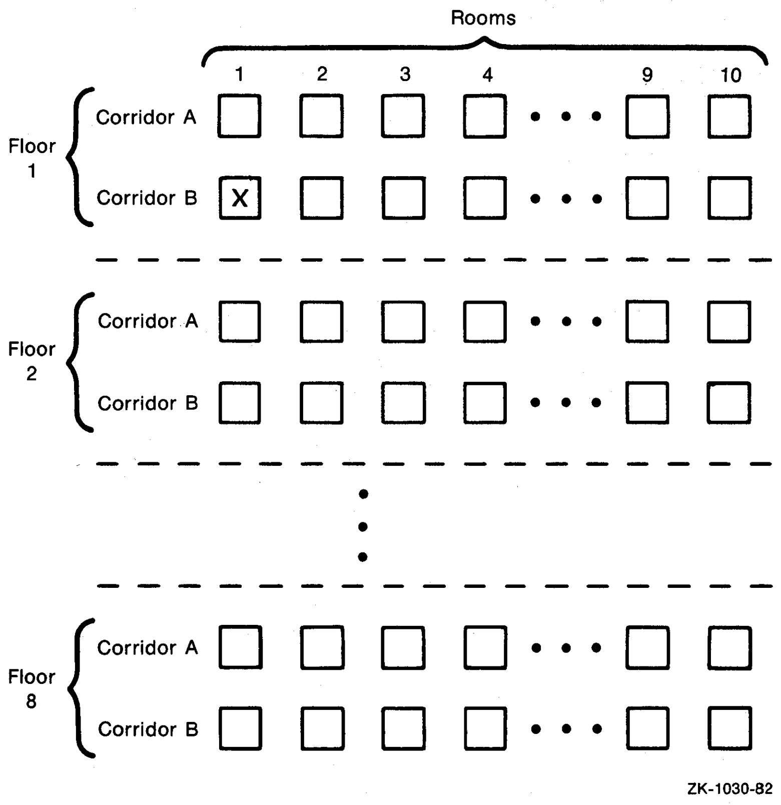

Figure 6–2: The Three-Dimensional Array Hotel_Vacancies

On the first floor, room number 1 on corridor B is denoted by Hotel_Vacancies[1,′B′,1]; that component is indicated in Figure 6–2 with an X. Note that one index type in Hotel_Vacancies is a subrange of type CHAR and the other two index types are subranges of type INTEGER.

6.1.2 Character Strings

A *string constant* is a sequence of characters enclosed in apostrophes. A *string variable* is defined as a one-dimensional *packed* array of characters.

6.1.2.1 Character-String Constants — Character-string constants are often used in output statements to print a message or a prompt. In addition, you can assign a string constant to a string variable if the variable is of the appropriate type.

The following are all character-string constants:

```
'Pay this amount -- $'
'Continual eloquence is tedious'          (* Blaise Pascal *)
'memorandum'
'365'
'The writer''s prerogative'
```

Note that the constant ′365′ is not an integer; it is a string of numeric characters. To indicate the apostrophe character in a string constant, you must type it twice, as in ′The writer′′s prerogative′.

6.1.2.2 Character-String Variables — The type of a character-string variable is a packed array of characters. Packing means that the characters are stored in memory as densely as possible. The index type must be an integer subrange with a lower bound of 1.

To pack an array, include the reserved word PACKED in the type definition. For instance, the following example defines a character-string type and declares a variable of that type:

```
TYPE
   String : PACKED ARRAY[1..20] OF CHAR;

VAR
   Name : String;
```

These declarations create a 20-component character-string variable called Name. Any character string assigned to Name must be exactly 20 characters long. PASCAL neither adds spaces to extend a shorter string nor truncates a longer string.

A string variable can be assigned the value of any string constant or variable of the correct length. For example, given the declarations of Chapter and Section shown below, you can assign string constants to each of them as shown.

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

VAR
   Chapter, Section : Title;
       .
       .
       .
Chapter := 'STRUCTURED TYPES    ';
Section := 'Character Strings   ';
```

You must include spaces at the end of each string so that each string contains 20 characters.

You can assign one string variable to another as follows:

```
Chapter := Section;
```

Both variables must be of the same type. Two string variables are of the same type if the upper bounds of their index types are the same.

You can use the READ or READLN procedure to read a sequence of characters into a string variable. For example, if Name is a variable of type String, that is, PACKED ARRAY[1..20] OF CHAR, you can write the procedure call:

```
READ (Name);
```

The READ procedure reads successive characters from a file into successive components of the array, starting with the component Name[1]. The read operation is complete when EOLN becomes TRUE or when the number of characters in the array (20 in Name) have been read. If EOLN becomes TRUE before a character is read into every component of the array, the remaining components are filled with spaces.

For example, if the READ (Name) procedure reads the following characters, up to the EOLN mark, from a file

```
Joshua Jones   EOLN
```

the value of the variable Name will be

```
'Joshua Jones        '
```

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 were additional characters on the line instead of the EOLN mark, those characters would have been 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 lexicographic 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. Remember that uppercase letters have lower ordinal values than lowercase letters.)

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 value of the string that contains the character with the higher ordinal value is greater than the value of the other string. If all characters are the same, including spaces, the values of the strings are equal.

For example, the following relational expressions are TRUE:

```
'programmers' < 'tech writers'
'wine & roses ' > 'wine & cheese'
```

The first expression is TRUE because the ordinal value of 'p' (112) is less than the ordinal value of 't' (116). When evaluating the second expression, PASCAL compares 'r' and 'c' because they 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!');
```

6.2 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 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 name of the record variable and the name of the field, separated by a period. Each field can be treated as a variable of the field type. For example:

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

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

The record type definition format is as follows:

```
  RECORD
{{field-name},...: type};...[[;]]
  END;
```

As shown, the type definition can specify the names of one or more fields, which can be of any type. If there are several fields of the same type, their names must be separated with commas. 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

The record declaration is one exception to the rule that every END must be associated with a BEGIN.

Suppose you are shopping for a new home and you want to maintain information on the houses you see. The important factors 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 identifier 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 identifier. For example:

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

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

```
FOR I := 1 TO Max_Houses DO
   BEGIN
   READLN (House_Choices[I].Cost);
   READLN (House_Choices[I].Distance);
   READLN (House_Choices[I].Num_Rooms);
   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 one record variable to another of the same type. For example, the following variable 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;
```

Records can be *nested* 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. For example:

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

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

When you are performing I/O operations on text files, you must read and write the information in a record field by field. 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, that is, Name and EmployeeNo.

When you refer to fields of the same record repetitively, 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},... 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 using 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 statement

```
READLN (Cost);
```

is the same as the statement

```
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, 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 := 239;
   END;
```

The record Hosp contains the field BirthDate, which is also a record (of type Date). By specifying Hosp in the WITH statement, you can refer to Patient and Age, which are fields of Hosp. By specifying BirthDate, you can 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 sequence of statements:

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

The record names in the WITH statement must be specified in the same 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 7
More PASCAL Statements

In addition to the statements presented in Chapter 4, PASCAL includes the following control statements:

- REPEAT statement
- WHILE statement
- CASE statement
- GOTO statement

This chapter describes the REPEAT, WHILE, and CASE statements, and gives a program example using various PASCAL statements. The GOTO statement, which transfers program control to a statement prefixed by a label, is explained in the *VAX–11 PASCAL Language Reference Manual.*

The REPEAT and WHILE statements, like the FOR statement, control loops. In both REPEAT and WHILE, the statement(s) within the loop body are executed repetitively depending on the value of a Boolean expression. The difference between REPEAT and WHILE lies in the point at which the value of the Boolean expression is tested.

The CASE statement is similar to the IF-THEN and IF-THEN-ELSE statements because it is a conditional statement. Remember that a conditional statement selects one statement for execution if a condition is met. Unlike IF-THEN and IF-THEN-ELSE, the CASE statement allows the selection to be made from a list of more than two statements.

7.1 The REPEAT Statement

The REPEAT statement allows you to specify the repetitive execution of a statement or series of statements until a certain condition becomes TRUE. The format is:

```
REPEAT {statement};...
UNTIL Boolean-expression
```

The statement(s) within the reserved words REPEAT and UNTIL can be any PASCAL statement(s). The loop body is executed until the Boolean expression becomes TRUE. The flow of control for the REPEAT statement is shown in Figure 7–1.

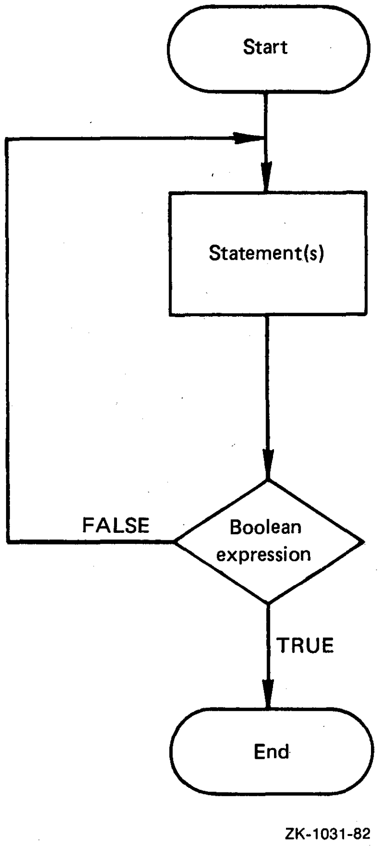

Figure 7-1: The REPEAT Statement Flow Chart

The example below shows the use of a REPEAT loop to search for a value in a sorted array. Assume that you have made 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, say, of authors. The following program fragment prompts for a name, then searches the array for that name.

```
(* Input the name of the author *)
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
      components 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 component', 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.');
```

The REPEAT loop contains statements that repetitively partition the array and search the appropriate half. The variables I and J initially represent the upper and lower bounds of the 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 the value of Size is 20, Name_to_Find is first compared 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 components 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 components Name_List[1] through Name_List[9].

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

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

Note the following properties of the REPEAT statement:

- The statement(s) in the loop body are always executed at least once because the Boolean expression in the UNTIL clause is evaluated after the loop body is executed.
- A statement or statements within the loop body must eventually cause the value of the Boolean expression to become TRUE. Otherwise, the loop would never stop executing.
- 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 may be delimited with BEGIN and END, but do not have to be. In addition, you need not use a semicolon immediately preceding UNTIL.

Examples

1. 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 Sum, and increases Count by 1.

   The REPEAT loop is terminated 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.

2. The following declarations have been made:

```
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 type PACKED ARRAY OF CHAR. Namelist can contain up to 30 names.

The REPEAT loop increases Namecount by 1, then reads one name and assigns it to one component of Namelist (that is, Namelist[Namecount]). The loop is terminated when Namecount equals 30 or when EOF becomes TRUE. Note that, 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.

7.2 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 body is executed while the Boolean expression is TRUE. When the expression becomes FALSE, execution terminates.

The flow of control for the WHILE statement is shown in Figure 7–2.

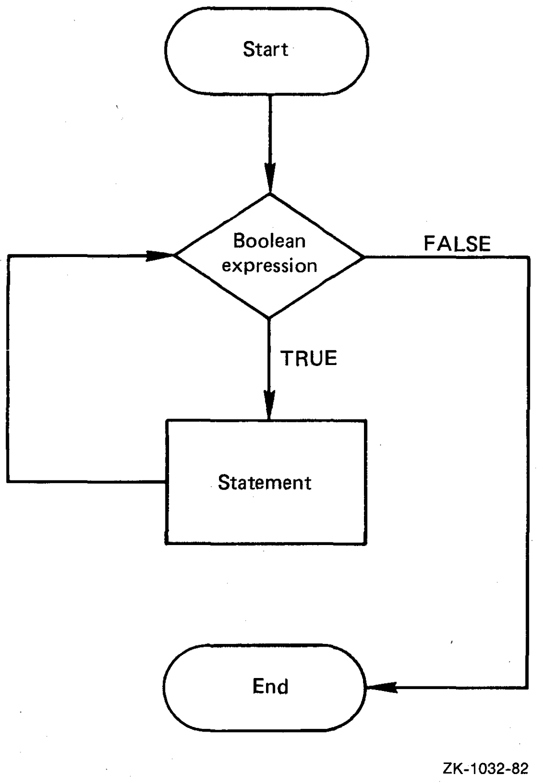

Figure 7–2: The WHILE Statement Flow Chart

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 body; 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 repetitively, you must use a compound statement. REPEAT does not require a compound statement.
3. WHILE terminates execution of the loop when a condition becomes FALSE. REPEAT terminates execution when the condition becomes TRUE.

Examples

1. 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 body is repeated as long as EOLN (INPUT) is FALSE. If the input line is empty, the WHILE loop is not executed at all, while the REPEAT loop is executed once. The REPEAT loop reads an additional line while searching for a number.
- The condition which determines that only 10 integers can be averaged must be rewritten as Count $<$ 10 (instead of Count $=$ 10). On the last

iteration, the 10th integer is read and Count becomes 10. Count < 10 is then FALSE, so the loop body 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.

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: if it is positive, the variable Poscount is incremented; if it 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 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 loop body will not be executed. Similarly, if Intcount is greater than or equal to 100, the loop will not be executed.

7.3 The CASE Statement

The CASE statement selects one out of a group of statements for execution. In a CASE statement, constant values, or *case labels*, are associated with each possible statement or action to be performed. CASE executes the statement labeled by the value that equals a specified expression. For example:

```
CASE 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 Answers to the case labels (the numbers 0 through 10). If the value of Answers is any of the numbers from 0 to 10, the statement to the right of that number is executed.

NOTE

The CASE statement is one exception to the rule that every END must be associated with a BEGIN.

The format of the CASE statement is:

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

The *case selector* is any expression (not only a variable) whose value is of an ordinal type. The case label list consists 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 may include an optional OTHERWISE clause that contains one or more statements that are not associated with labels.

You can specify the labels in any order. Each label may 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 statements following OTHERWISE are executed. It is an error if the selector does not equal a label and there is no OTHERWISE clause.

The flow of control for the CASE statement is shown in Figure 7-3.

Examples

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 will be assigned to the variable Season.

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);
```

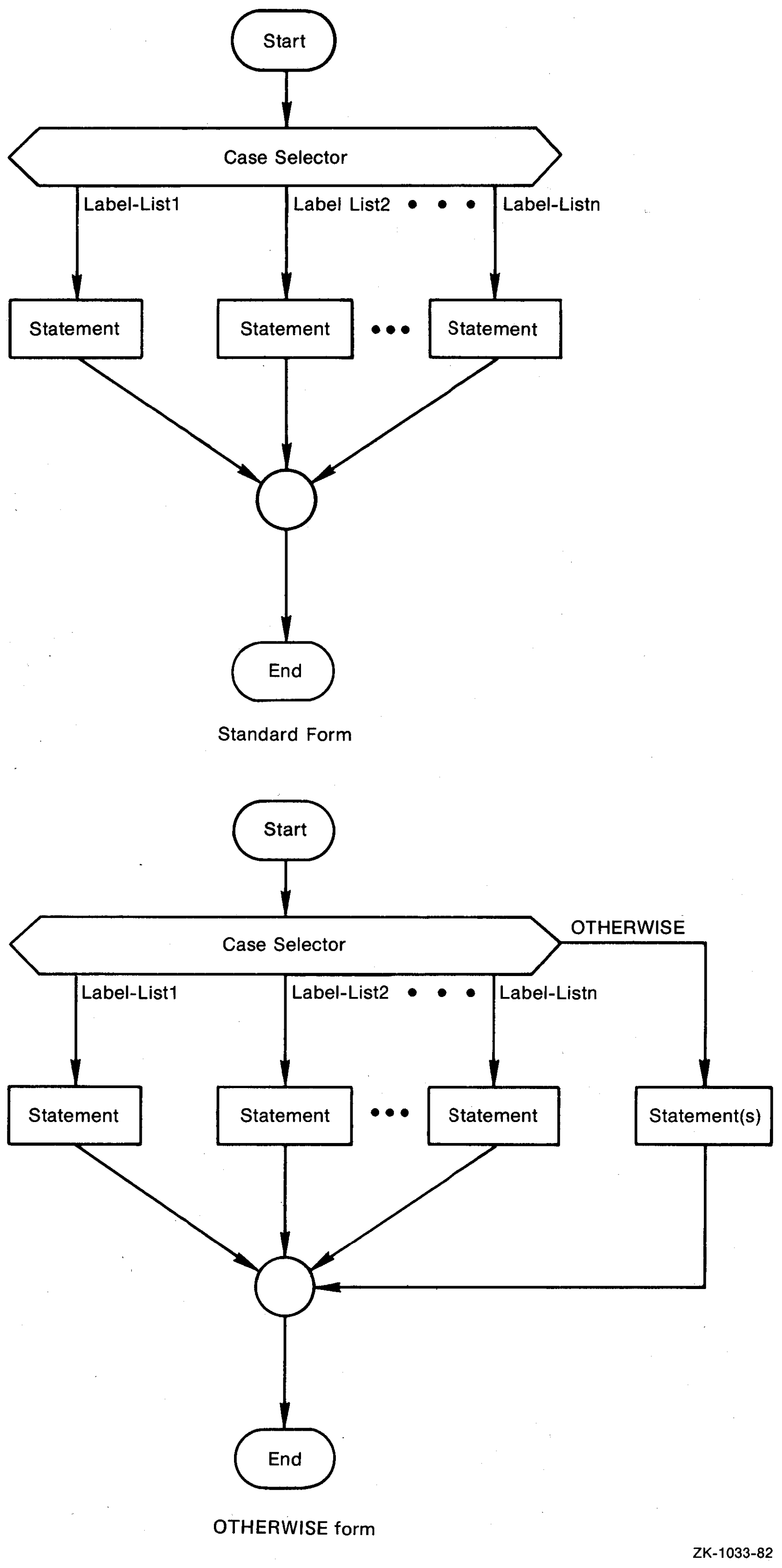

Figure 7–3: The CASE Statement Flow Charts

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.

7.4 The Program Class_Data — An Example

This section presents an example program called Class_Data. The program illustrates several of the language features covered in Chapters 5, 6, and 7.

The program Class_Data, illustrated in Figure 7-4, reads and stores information about a hypothetical group of students. The data for one student is stored in a variable (Student) whose type is a record with three fields. The three fields contain the following information:

- Social security number
- Name
- Year

The array Class contains information about a group of students. The executable section illustrates the use of the WITH, WHILE, and CASE statements, I/O procedures, and various other PASCAL features.

```
PROGRAM Class_Data (INPUT,OUTPUT);

(* Declarations *)

CONST
   Max_Size = 100;❹

TYPE
   Student_Id = ❶
      RECORD
      SocSec : PACKED ARRAY[1..11] OF CHAR;  (* SocSec is in the form: ###-##-#### *)
      Name : PACKED ARRAY[1..20] OF CHAR;
      Year : (Freshman, Sophomore, Junior, Senior);
      END;

VAR
   Student : Student_Id;❷
   Class : ARRAY[1..Max_Size] OF Student_Id;❸
   ClassSize, I : INTEGER := 0;
   Fresh_Count, Soph_Count, Jun_Count,  ❺ (* Initializes each of these integer *)
   Sen_Count : INTEGER := 0;               (* variables to 0 *)

  BEGIN       (* Class_Data *)
  (* Instructions *)
❻ WRITELN ('For each student, type a name, soc. sec. #, and year, as');
  WRITELN ('prompted. Press <RET> after each answer and <CTRL/Z> after list');
  WRITELN;
```

Figure 7-4: The Program Class_Data

```
  (* Input Section *)
  WRITE ('Name: ');                                 (* Initial prompt *)

  (* This loop prompts for and then reads data for a student record, assigns
     it to one component of the array Class, and increments the array index
     and the variable ClassSize. The loop is terminated when EOF becomes true
     (that is, when <CTRL/Z> is encountered.) *)

  WHILE NOT EOF DO 10
     BEGIN
     WITH Student DO 11
        BEGIN
        READLN (Name);
7       WRITE ('Soc Sec #: '); 13
        READLN (SocSec); 14
        WRITE ('Year: ');
        READLN (Year);
        END;
8    ClassSize := ClassSize + 1;
     Class[ClassSize] := Student;
     IF ClassSize < (Max_Size)
     THEN
        WRITE ('Name: ')
     ELSE
        WRITELN ('The class is full, type <CTRL/Z>');
     END;
  WRITELN;

  (* Output Section *)
  (* The following section prints the output data. The output consists of a
     heading, and a name, number, and year for each student. In addition, the
     number of students in each year is counted and reported in the output. *)

  WRITELN ('Name:', 'SocSec #:':24, 'Year:':14);
  WRITELN;
  FOR I := 1 TO ClassSize DO 12
     BEGIN                                                16
        WITH Class[I] DO
           BEGIN
           WRITELN (Name:20, SocSec:11, Year:12);
9          CASE Year OF ------------------------------ 15
                Freshman : Fresh_Count := Fresh_Count + 1;
                Sophomore : Soph_Count := Soph_Count + 1;
                Junior : Jun_Count := Jun_Count + 1;
                Senior : Sen_Count := Sen_Count + 1;
           END;
        END;
     END;
  WRITELN;
  WRITELN ('Class Profile:');
  WRITE (Fresh_Count:2, ' freshmen ', Soph_Count:2, ' sophomores ');  16
  WRITELN (Jun_Count:2, ' juniors ', Sen_Count:2, ' seniors ');
  END.           (* Class_Data *)
                                                    17
  (* Sample Output
  Name:                SocSec #:      Year:

  Kathy Moore          234-34-5678   FRESHMAN
  John Jones           345-67-8907     SENIOR
  Dave Brown           078-34-2345     JUNIOR
  Ruth Doe             121-21-2121   FRESHMAN
  Barb Cohen           000-00-0000     SENIOR
  Roy Rogers           234-56-7890  SOPHOMORE

  Class Profile:
   2 freshmen  1 sophomores  1 juniors  2 seniors   *)
```

Figure 7-4 (Cont.): The Program Class_Data

The circled numbers in Figure 7–4 are keyed to the numbers in the following sections.

7.4.1 The Declaration Section

The major data structure in the program Class_Data is the user-defined type Student_Id ➊. Student_Id is a record containing three fields — SocSec, Name, and Year. The field SocSec is a packed array of 11 characters. Social security numbers are to be input in this format:

```
xxx-xx-xxxx
```

The field Name is a packed array of 20 characters. The field Year is an enumerated type consisting of the four class levels — Freshman, Sophomore, Junior, and Senior.

The variable Student ➋ is declared to be of type Student_Id. It can contain only one data record (that is, one name, social security number, and year). The variable Class ➌ can contain a group of records. Class is an array with components of type Student_Id and with indexes that range from 1 to Max_Size. Max_Size ➍ is a symbolic constant defined as 100 in the CONST section. This CONST definition can be easily modified to accommodate a larger or smaller group of students.

The remaining variables are integers ➎ that represent the number of students in the entire class, the number of students in each year, and a FOR loop control variable (the variable I). Each of the count variables is initialized to 0 when it is declared.

7.4.2 The Executable Section

The program Class_Data accepts data on a group of students, stores the information in an array of records, and prints the information. The program follows the steps outlined below:

- Prints instructions for the user ➏.
- Prompts for and then reads input data for each student ➐.
- Increments the index variable ClassSize and assigns data for one student to one component of the array Class ➑.
- Prints collected data, including headings ➒. The output data consists of a name, a social security number, and a year for each student, and the number of students in each of the four years.

The executable section contains various PASCAL statements and I/O procedures. For example, the WHILE loop ➓ determines the flow of control for the section that prompts for and reads data. The statements within the WHILE loop are executed repetitively as long as NOT EOF is TRUE.

There are two examples of the WITH statement in Class_Data ⓫. The first example is:

```
WITH Student DO
```

This statement allows direct references to fields of the record variable Student in the subsequent compound statement. Thus, the following statements are equivalent:

```
READLN (Name);                    READLN (Student.Name);
```

The WITH statement is also used in the output section. In this WITH statement, the specified record variable is one component of the array Class. Because the FOR loop ⓬ increments the index variable I, a different component of Class is processed each time the WITH statement is executed.

In addition, the program illustrates two extended I/O features of VAX-11 PASCAL: (1) prompting at the terminal and (2) reading of character strings. The input section contains the following statement, which prompts for input data at the terminal ⓭:

```
WRITE ('SocSec #: ');
```

This statement prints the string enclosed in apostrophes and leaves the carriage or cursor positioned after the last character printed. You can then finish the line by typing a string that represents a social security number:

```
SocSec #: 425-29-0000
```

The following statement ⓮ reads the string into a packed array of characters:

```
READLN (SocSec);
```

Each character in the social security number (including hyphens) is assigned to a component of Student.SocSec.

Class_Data shows the use of a CASE statement ⓯ in the output section. The selector in the CASE statement is Year, which is one field of the current component of Class (that is, Class[I]). As the FOR loop steps through each component of Class, this CASE statement counts the number of occurrences of each year. For example, if Class[I].Year equals Sophomore, the CASE statement selects the statement to the right of the value Sophomore. Thus, Soph_Count is increased by 1.

The WRITE and WRITELN statements in the output section ⓰ illustrate various examples of field-width specifications. These statements produce the sample output ⓱ shown below the program.

Chapter 8
Procedures and Functions

In many cases, it is convenient to group into a discrete unit statements that perform some specific action in a program. In PASCAL, procedures and functions are examples of such discrete units. You can write procedures and functions to perform specific tasks. For example, all of the output operations needed in a program can be contained in one procedure. The program can call the procedure every time the output operations are needed.

A *procedure* is a named group of statements that performs a set of actions. You declare a procedure in the declaration section. Subsequently, you can call the procedure in the executable section. When a procedure is called, the statements are executed as a group.

A *function* is similar to a procedure in that (1) it is a named set of statements, (2) you must declare it in the declaration section, and (3) the statements are executed as a group when the function is called by a *function designator*. However, a function has a type and returns a value of that type. You can use a function designator just as you would use any expression; in fact, it is an expression.

Procedures and functions have similar structures and restrictions. This primer uses the term *routine* in descriptions that apply to both procedures and functions. You can use *predeclared routines*, which are declared by PASCAL and denoted by predeclared identifiers, or you can create user-declared routines, as described in this chapter. Appendix C contains tables of all the predeclared procedures and functions in VAX–11 PASCAL.

The sample program Compute, shown in Figure 8–1, illustrates some general concepts that apply to all routines. The explanations that follow Figure 8–1 are keyed to the figure by means of the circled numbers.

Sections 8.1 and 8.2 describe procedures and functions, respectively, in detail. Section 8.3 discusses how to pass data, in the form of *parameters*, to a routine.

```
PROGRAM Compute (INPUT, OUTPUT);

(* This program computes the minimum, maximum, and average values in a list of
   integers typed at the terminal and the number of times the minimum and maximum values
   occur. The number of integers it accepts is determined by the symbolic constant Number. *)

CONST
   Number = 25;                          (* Max number of values to be processed *)

TYPE
   Range = 0..1000;                      (* Values can be in this range *)
   List = ARRAY[1..Number] OF Range;     (* Specifies the amount and range for values *)

VAR
   Arr : List;                           (* Holds the values to be processed *)
   Minimum, Maximum : Range;             (* Minimum and Maximum of list *)
   Average : REAL;                       (* Average value in list *)
   Int_Count : INTEGER;                  (* Number of values read from terminal *)

PROCEDURE Read_Ints ❹
   (VAR A : List;     }
    VAR I : INTEGER); } ❽ ❿

(* This procedure reads the integers to be processed into the array A. The
   array of integers and the number of integers that were read are passed back
   to the main program. *)

   BEGIN
   WRITELN ('Type from 1 to ', Number:3, ' integers, in the range of 0 to 1000.');
   WRITELN ('Type <RET> after each integer and <CTRL/Z> after last integer.');
   I := 0;
   WHILE NOT EOF (INPUT) AND (I < Number) DO
     BEGIN
     I := I + 1;
     READLN (A[I]);
     END;
   END;                                  (* Read_Ints *)

PROCEDURE Min_Max_Avg ❺
   (A : List;                }
    Ints_Read : INTEGER);    } ❽

(* This procedure computes the minimum, maximum, and average values in array A.
   It also counts the occurrences of the minimum and maximum values; Each of these
   computations is printed with text that labels each value. *)

  {VAR
  {   Sum, J : INTEGER;
❾{   Max_Count, Min_Count : 1..Number;
  {   Min, Max : Range;
  {   Avg : REAL;
```

Figure 8–1: The Program Compute

```
PROCEDURE Print_Data; ❻
   BEGIN
   WRITELN;
   WRITELN ('Maximum =', Max:4,', occurring', Max_Count:4, ' times');
   WRITELN ('Minimum =', Min:4,', occurring', Min_Count:4, ' times');
   WRITELN ('Average value (truncated) =', TRUNC(Avg):6);
   WRITELN ('Average value =', Avg : 12);
   END;

BEGIN                                    (* Min_Max_Avg *)
Max := A[1];
Min := Max;
Sum := Max;
Max_Count := 1;
Min_Count := 1;
(* Begin following FOR loop with a 2 because Max and Min
   already contain the first component in array. The first
   iteration compares the second component to the first component. *)
FOR J := 2 TO Ints_Read DO               (* A[J] is current integer in array *)
   BEGIN
   Sum := Sum + A[J];
   IF A[J] > Max                         (* If A[J] > Max, assign it to Max *)
   THEN
      BEGIN
      Max := A[J];
      Max_Count := 1;
      END
   ELSE
      IF A[J] = Max                      (* If A[J] = Max, increment Max_Count *)
      THEN
         Max_Count := Max_Count + 1;
   IF A[J] < Min                         (* If A[J] < Min, assign it to Min *)
   THEN
      BEGIN
      Min := A[J];
      Min_Count := 1;
      END
   ELSE
      IF A[J] = Min                      (* If A[J] = Min, increment Min_Count *)
      THEN
         Min_Count := Min_Count + 1;
   END;
   Avg := Sum/Ints_Read;
   Print_Data; ❼                         (* Print results *)
   END;                                  (* Min_Max_Avg *)

(****** MAIN PROGRAM ******)
                       ⓫
  BEGIN
❶ Read_Ints (Arr, Int_Count); ❷
  Min_Max_Avg (Arr, Int_Count); ❸
  END.
```

Figure 8-1: (Cont.) The Program Compute

The program Compute finds the minimum, maximum, and average values in a list of integers typed at the terminal.

Execution starts with the BEGIN that delimits the executable section of the main program ❶ and follows these steps:

1. The main program calls the procedure Read_Ints ❷, which performs these steps:

   a. Types instructions to the terminal

   b. Reads integers and assigns them to successive components of the array A, terminating when EOF is TRUE or when 25 values have been read

2. The main program calls the procedure Min_Max_Avg ❸, which performs these steps:

   a. Finds the minimum and maximum values of the integers stored in the array, and computes the number of times each minimum and maximum occurred

   b. Computes the average number of integers read

   c. Calls the procedure Print_Data, which prints the result of each computation with text that labels each value

Format of a Routine

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

Thus, in Figure 8–1, the procedures Read_Ints ❹ and Min_Max_Avg ❺ appear in the declaration section of the program Compute. These procedures are executed as a result of *procedure calls* ❷ ❸, which appear in the executable section of Compute.

Routines are said to be *nested* within the main program. In addition, routines can be nested within other routines. You can refer to the name of a nested routine only from inside the block that declares it. For example, the procedure Print_Data is nested within the procedure Min_Max_Avg ❻. Print_Data is executed as a result of a procedure call ❼ in the body of Min_Max_Avg.

Routines are similar in format to programs. A routine consists of a heading and a block; the block contains a declaration section and an executable section. The heading of a routine is slightly different from that of a program: it can contain a *formal parameter list* ❽. The heading of a function also indicates the type of the value returned. The declaration section defines local data items that are used in the routine. The executable section contains the statements that perform the actions of the routine.

Identifiers Used in Routines

The statements in a routine can use three kinds of user identifiers:

- Local identifiers
- Global identifiers
- Parameters

Local identifiers are defined in the declaration section of a routine. They can be accessed only from within the routine block in which they are declared. For example, the procedure Min_Max_Avg uses the local identifiers Sum, J, Max_Count, Min_Count, Min, Max, Avg, and Print_Data ❾. The procedures Read_Ints and Print_Data use no local identifiers.

Global identifiers are declared outside a routine; they can be accessed by the routine. Thus, identifiers declared in the main program block are global to all routines. Identifiers declared in a routine are global to all nested routines. For example, the procedure Print_Data has no local variables. It uses the global identifiers defined in the surrounding block Min_Max_Avg ❾. In addition, predeclared identifiers are global to all parts of a PASCAL program.

Parameters are the means by which routines can communicate with the main program and with each other. The parameters in the routine heading are called *formal parameters.* Their identifiers are used within the routine block. Each formal parameter corresponds to an *actual parameter* found in the routine call. During execution, the formal parameters within the routine take on the values of the actual parameters.

For example, in the procedure Read_Ints, the formal parameter list declares the parameters A and I of type List and INTEGER, respectively ❿. The procedure is called with two actual parameters: Arr of type List and Int_Count of type INTEGER ⓫. The identifiers A and I are used within Read_Ints to represent the variables Arr and Int_Count.

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 an identifier declared in the main program block is the full program. The scope of an identifier declared in a routine block is that routine and all routines nested within it.

In a routine, you can redeclare an identifier that has been declared in an outer block. When you use an identifier that is declared both in a routine and in an outer block, the identifier always refers to the declaration of most limited scope. The scope of the global identifier does not include any block in which it is redeclared. Thus, the local identifier can have properties (for instance, type or value) distinct from those of the global identifier.

8.1 Procedures

A *procedure* associates an identifier with a group of statements. For example, the program Compute contains a simple procedure called Print_Data that performs output operations:

```
PROCEDURE Print_Data;
   BEGIN
   WRITELN;
   WRITELN ('Maximum =', Max:4, ', occurring', Max_Count:4, ' times');
   WRITELN ('Minimum =', Min:4, ', occurring', Min_Count:4, ' times');
   WRITELN ('Average value (truncated) =', TRUNC (Avg):6);
   WRITELN ('Average value =', Avg:12);
   END;
```

The specification of Print_Data in the procedure Min_Max_Avg is a procedure declaration.

8.1.1 Declaring a Procedure

To declare a procedure, specify the *procedure heading* and block in a PROCEDURE part of the declaration section. You can declare a procedure in the main program or in another routine. The format of a procedure declaration is:

```
PROCEDURE procedure-name ‖(formal-parameter-list)‖;
   ‖declaration-section‖
   BEGIN
   {statement};...
   END;
```

The procedure name specifies the identifier to be used as the name of the procedure. The formal parameter list describes the formal parameters used in the procedure (see Section 8.3).

The declaration section contains local declarations and definitions. The statements between BEGIN and END can be any PASCAL statements. In short, the procedure block is identical to the main program block with the following exceptions:

- The declaration section cannot usually contain value initializations.
- The procedure block ends with an END followed by a semicolon (;) rather than by a period (.).

You cannot redeclare the formal parameter names as local identifiers in the procedure.

8.1.2 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.

For example, to execute the procedure Print_Data, you specify its name as a procedure call:

```
Print_Data;
```

Note that the procedure call is itself a statement.

The main program in Compute (Figure 8–1) consists of two statements — a procedure call to Read_Ints and a procedure call to Min_Max_Avg:

```
BEGIN
Read_Ints (Arr, Int_Count);
Min_Max_Avg (Arr, Int_Count);
END.
```

The call to Read_Ints specifies two actual parameters, the variables Arr and Int_Count. They correspond to the formal parameters A and I in the procedure. The call to Min_Max_Avg passes the same two variables as actual parameters. In Min_Max_Avg, the formal parameters A and Ints_Read take on the values of the actual parameters Arr and Int_Count.

Actual and formal parameters are described in greater detail in Section 8.3.1.

8.2 Functions

A *function* associates an identifier with a group of statements that returns a value. 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 *return value* is specified in the *function heading*.

For example, the following is a valid function declaration:

```
FUNCTION Divides
   (Dividend, Divisor : INTEGER)
   : BOOLEAN;

   BEGIN
   IF (Dividend MOD 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, if Divisor can be divided into Dividend with a zero remainder), the function Divides will be TRUE. Otherwise, the function Divides will be FALSE.

8.2.1 Declaring a Function

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

```
FUNCTION function-name ‖(formal-parameter-list)‖ : result-type;
    ‖declaration-section‖
    BEGIN
    {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 8.3.1. The result type may be any type except a file type or a structured type that has a file component.

The block of a function is the same as the block of a procedure. Remember that routines cannot usually contain value initializations and that they are terminated with a semicolon (;) instead of a period (.).

Every function must include at least one statement that assigns a value of the result type to the function name. In the function Divides, if the Boolean expression — (Dividend MOD Divisor = 0) — is TRUE, the value TRUE will be assigned to the function name. Otherwise, FALSE will be assigned to the function name. If a value is not assigned to the function name during execution of the function, the function result will be undefined.

8.2.2 Invoking a Function

A function is called, and thereby executed, as a result of a *function designator.* A function designator is an expression and thus can be used wherever other expressions can be used. For example, a function designator may appear on the right-hand side of an assignment statement.

You can use the following statement to call the function Divides:

```
IF Divides (Num1, Num2)
THEN
   WRITELN (Num1:4, ' is a multiple of ', Num2:4)
ELSE
   WRITELN (Num1:4,' is not a multiple of ', Num2:4);
```

In this statement, the following expression is a function designator:

```
Divides (Num1, Num2)
```

The function designator is a Boolean expression in the IF-THEN-ELSE statement. As with a procedure, the actual parameters in a function designator are passed to the function. Thus, in the function Divides, the parameters Dividend and Divisor take the values of Num1 and Num2, respectively.

If, as a result of the IF-THEN-ELSE statement, Divides returns TRUE, the first WRITELN procedure will be performed. If Divides returns FALSE, the second WRITELN procedure will be performed.

Actual parameters are used in a function designator in exactly the same manner as they are used in procedure calls. See Section 8.3.1 for details.

8.3 Parameters

You pass data (that is, values and variables) to a routine by means of *parameters*. Formal parameters are those listed in the routine declaration. Actual parameters are those specified in the routine call. In the body of the routine, the formal parameters represent the actual parameters.

VAX–11 PASCAL contains several extensions to the syntax of formal and actual parameters. The *VAX–11 PASCAL Language Reference Manual* provides complete information on these extensions.

8.3.1 Actual and Formal Parameters

A routine's formal parameters assume the values of the actual parameters when the routine is called. Formal parameters are listed in the *formal parameter list* in the heading of the routine. The formal parameter list describes the parameters used within the routine and their types. The formal parameter list is the declaration for the routine's parameters; they are not declared elsewhere. For example, a procedure named Print_Sym_Array may have the following heading:

```
PROCEDURE Print_Sym_Array
   (VAR A : Arr;
    Side : INTEGER);
```

In this procedure, A and Side are formal parameters and their types are Arr and INTEGER, respectively.

The format of the formal parameter list for specifying value or variable parameters (see Section 8.3.2) is:

(⟦VAR⟧ {identifier-list : type};...)

The reserved word VAR is placed only before variable parameters. The identifier list specifies one or more identifiers that denote formal parameters. The type specifies the type of the parameters in the preceding identifier list.

The formal parameter list determines the contents of the *actual parameter list*. In the actual parameter list, you include one actual parameter for each formal parameter specified in the routine heading. All identifiers used in the actual parameter list must be declared in the block surrounding the routine call.

For example, a valid procedure call to the procedure Print_Sym_Array is:

```
   Print_Sym_Array (Current_Arr, Current_Side);
```

The actual parameter Current_Arr is associated with the formal parameter A, and the actual parameter Current_Side is associated with the formal parameter Side.

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

You can call a routine several times with different actual parameters. For example, the same program that contains the procedure call to Print_Sym_Array (shown above), can contain the following:

```
Print_Sym_Array (Another_Arr, Another_Side);
```

As a result of this procedure call, when Print_Sym_Array is executed, the formal parameters A and Side represent the actual parameters Another_Arr and Another_Side, respectively.

To illustrate further the use of parameters, the procedure Print_Data from Figure 8–1 can be rewritten so that it uses parameters instead of global variables:

```
PROCEDURE Print_Data
   (PMax, PMin : Range;
    PMax_Count, PMin_Count : INTEGER;
    PAvg : REAL);

    BEGIN
    WRITELN;
    WRITELN ('Maximum =', PMax:4, ', occurring', PMax_Count:4, ' times');
    WRITELN ('Minimum =', PMin:4, ', occurring', PMin_Count:4, ' times');
    WRITELN ('Average value (truncated) =', TRUNC (PAvg):6);
    WRITELN ('Average value =', PAvg:12);
    END;
```

The formal parameter list describes the variables used in the procedure and their types. PMax and PMin are parameters of type Range (defined as 0..1000 in the program Compute). PMax_Count and PMin_Count are parameters of type INTEGER, and PAvg is a parameter of type REAL.

The following is a valid procedure call to the rewritten procedure Print_Data:

```
Print_Data (Max, Min, Max_Count, Min_Count, Avg);
```

Within the procedure, the formal parameters are used in the WRITELN procedures as if they already had values, even though no values are assigned to them. When the procedure is executed, each formal parameter (PMax, PMin, PMax_Count, PMin_Count, and Avg) assumes the value of its corresponding actual parameter (Max, Min, Max_Count, Min_Count, and Avg, respectively).

The identifiers Max, Min, Max_Count, Min_Count, and Avg must be declared outside the procedure Print_Data. (In Compute, they are declared in the declaration section of the procedure Min_Max_Avg.) Each of these parameters must have the same type as its corresponding formal parameter.

8.3.2 Value and Variable Parameters

You can specify several kinds of parameters in a routine's formal parameter list. The kind of parameter specified in the routine heading determines how the parameter is passed to the routine. The two most common kinds of parameters are:

1. *Value parameters* — the value of the actual parameter is made available to the routine. The routine cannot change the actual parameter's value during execution.
2. *Variable parameters* — the *address* of the actual parameter variable is made available to the routine. The routine can change the actual parameter's value.

PASCAL passes value parameters to routines by default. However, if you include the reserved word VAR before a parameter in the formal parameter list, the corresponding actual parameter will be passed as a variable parameter.

8.3.2.1 Value Parameters — When you do not want a routine to change the value of an actual parameter, you pass the actual parameter as a value parameter. An actual parameter passed as a value parameter can be any expression that can be assigned to the formal parameter type.

The procedure Min_Max_Avg in the program Compute specifies only value parameters in its heading:

```
PROCEDURE Min_Max_Avg
   (A : List;
    Ints_Read : INTEGER);
```

This declaration specifies that the parameters in the procedure call will be passed as value parameters. The procedure call is:

```
Min_Max_Avg (Arr, Int_Count);
```

When the procedure is executed, it uses the values of the variables Arr and Int_Count whenever the parameters A and Ints_Read are specified. Even if the procedure changed the values of A and Ints_Read, the values of Arr and Int_Count would be unchanged.

An actual value parameter can be an expression. For example, you could use the following procedure call to Min_Max_Avg:

```
Min_Max_Avg (Arr, Int_Count - Extras);
```

When the call is executed, the expression Int_Count - Extras will be evaluated and the resulting value passed to the formal parameter Ints_Read.

8.3.2.2 Variable Parameters — When you want a routine to change the value of an actual parameter, you must pass the actual parameter as a variable parameter.

To specify a variable parameter, you must include the reserved word VAR before a list of identifiers in the formal parameter list. VAR can appear more than once in the routine heading. An actual parameter that is passed as a variable parameter must be a variable of the same type as the corresponding formal parameter.

For example, the procedure Read_Ints declares variable parameters in its formal parameter list:

```
PROCEDURE Read_Ints
   (VAR A : List;
    VAR I : INTEGER);
```

The corresponding actual parameters must be variables of type List and INTEGER. Thus, the following is a valid procedure call:

```
Read_Ints (Arr, Int_Count);
```

Note that only the variable names appear in the actual parameter list; the reserved word VAR is not included.

When you use a variable parameter, the memory address of the actual parameter is made available to the routine. Therefore, in the example above, the formal parameter A and the actual parameter Arr represent the same variable. When the routine changes the value of A[I], it actually changes the value of Arr[I].

The routine Read_Ints contains the following statements:

```
WHILE NOT EOF (INPUT) AND (I < Number) DO
   BEGIN
   I := I + 1;
   READLN (A[I]);
   END;
```

The WHILE loop assigns values to components of array A until EOF is TRUE or until I is equal to Number. After the routine is executed, the array Arr and the variable Int_Count reflect the values assigned to A and I in the routine.

The procedure Read_Ints illustrates the importance of VAR parameters. The main purpose of Read_Ints is to read integers from the terminal into an array. If you use value parameters, the integers that are read will be lost after the procedure is executed. However, if you use VAR parameters, the values in the array and the number of integers that were read are reflected in the main program. The next line in the program uses these new values (Arr and Int_Count) in the procedure call to Min_Max_Avg.

You can have both value and variable parameters in the same formal parameter list. For example:

```
FUNCTION Symmetry
   (VAR SymArr : Square_Arr;
    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 will return 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-component array of integers. The parameter Side is of type INTEGER and holds the length of the sides of the 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');
```

Suppose 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.

Appendix A
PASCAL-Defined Names

This appendix contains lists of the names defined by VAX–11 PASCAL. Section A.1 lists the standard reserved words that cannot be redefined as identifiers. Section A.2 lists nonstandard reserved words that VAX–11 PASCAL allows you to redefine. 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 Standard Reserved Words

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

A.2 Nonstandard Reserved Words

%DESCR	MODULE
%IMMED	OTHERWISE
%INCLUDE	REM
%REF	VALUE
%STDESCR	VARYING

A.3 Predeclared Identifiers

ABS
ADD_INTERLOCKED
ADDRESS
ARCTAN
BIN
BITNEXT
BITSIZE
BOOLEAN
CARD
CHAR
CHR
CLEAR_INTERLOCKED
CLOCK
CLOSE
COS
DATE
DBLE
DELETE
DISPOSE
DOUBLE
EOF
EOLN
ESTABLISH
EXP
EXPO
FALSE
FIND
FINDK
GET
HALT
HEX
INDEX
INPUT
INT
INTEGER
LENGTH
LINELIMIT
LN
LOCATE
LOWER
MAXINT
NEW
NEXT
NIL
OCT
ODD
OPEN
ORD
OUTPUT
PACK
PAD
PAGE
PRED
PUT
QUAD
QUADRUPLE
READ
READLN
READV
REAL
RESET
RESETK
REVERT
REWRITE
ROUND
SET_INTERLOCKED
SIN
SINGLE
SIZE
SNGL
SQR
SQRT
STATUS
SUBSTR
SUCC
TEXT
TIME
TRUE
TRUNC
TRUNCATE
UAND
UFB
UINT
UNDEFINED
UNLOCK
UNOT
UNPACK
UNSIGNED
UOR
UPDATE
UPPER
UROUND
UTRUNC
UXOR
WRITE
WRITELN
WRITEV

Appendix C
Summary of Predeclared Procedures and Functions

Tables C–1 and C–2 summarize the procedures and functions declared by VAX–11 PASCAL. Some of the procedures and functions listed are not described in this primer. Some procedures have an optional parameter, described as e in this appendix. The value of this parameter indicates how errors should be handled if they occur during execution of the procedure. For further information, see the *VAX–11 PASCAL Language Reference Manual.*

Table C-1: Predeclared Procedures

Procedure	Parameter	Action

CLOSE(f,parameters,e)	f = file variable
	parameters — see the *VAX-11 PASCAL Language Reference Manual*
	e = error parameter
		Closes file f with the specified properties.

DATE(str)	str = variable of type PACKED ARRAY [1..11] OF CHAR	Assigns current date to str.

DELETE(f,e)	f = file variable
	e = error parameter
		Deletes current component of file f. File f must have relative or indexed organization and be opened for direct or keyed access. The current component must be locked.

DISPOSE(p)	p = pointer value	Releases storage for p^. Any pointers to the storage become undefined.
DISPOSE(p, t1,...,tn)	p = pointer value t1,...,tn = tag field constants	Releases 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.
ESTABLISH(id)	id = function-identifier	Sets up a VAX-11 condition handler to process exceptions.

FIND(f,n,e)	f = file variable
	n = component number
	e = error parameter
		Moves the current file position to component n of file f.

FINDK(f,kn,kv,m,e)	f = file variable
	kn = key number
	kv = key value
	m = match type
	e = error parameter
		Moves the current position of file f to a specified component. The match type can have a value of EQL, GTR, or GEQ to indicate that the component to be found has a value in key position kn that is equal to, greater than, or greater than or equal to key value kv. Match type m is optional and defaults to EQL. File f must be opened for keyed access.

GET(f,e)	f = file variable
	e = error parameter
		Moves the current file position to the next component of f. Then GET(f) assigns the value of that component to f^, the file buffer variable.

HALT	None	Calls LIB$STOP, signaling PAS$_ABORT. Without an appropriate condition handler, HALT terminates execution of the program.

LOCATE(f,n,e)	f = file variable
	n = component number
	e = error parameter
		Positions file f at component n so that the next PUT procedure can modify n.

LINELIMIT(f,n,e)	f = text file variable
	n = integer expression
	e = error parameter
		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.

NEW(p)	p = pointer variable	Allocates storage for p^ and assigns its address to p.
NEW(p, t1,...,tn)	p = pointer variable t1,...,tn = tag field constants	Allocates storage for p^; used when p^ is a record with variants. The optional parameters t1 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,parameters,e)	f = file variable
	parameters — see the *VAX-11 PASCAL Language Reference Manual*
	e = error parameter
		Opens file f with the specified properties.

(Continued on next page)

Table C-1: (Cont.) Predeclared Procedures

Procedure	Parameter	Action

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

PAGE(f,e)	f = text file variable
	e = error parameter
		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,e)	f = file variable
	e = error parameter
		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,e)	f = file variable
	v1,...,vn = variables
	e = error parameter
		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,e)	f = text file variable
	v1,...,vn = variables
	e = error parameter
		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.

READV(s,v1,...,vn)	s = character-string expression v1,...,vn = variables	Assigns successive values from the input string s to the variables v1 through vn. You must specify at least one variable (v1).

RESET(f,e)	f = file variable
	e = error parameter
		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.

RESETK(f,kn,e)	f = file variable
	kn = key number
	e = error parameter
		Enables reading from file f. RESETK(f,kn) moves the current file position to the component with the lowest value in key position kn. File f must be opened for keyed access.

REVERT	None	Cancels a VAX-11 condition handler set up by ESTABLISH.

REWRITE(f,e)	f = file variable
	e = error parameter
		Enables writing to file f. REWRITE(f) truncates the file f to zero length and sets EOF(f) to TRUE.

TIME(str)	str = variable of type PACKED ARRAY [1..11] OF CHAR	Assigns the current time to str.

TRUNCATE(f,e)	f = file variable
	e = error parameter
		Deletes current file component and all components following it. File f must have sequential organization.

UNLOCK(f,e)	f = file variable
	e = error parameter
		Unlocks the current file component if it is locked.

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

UPDATE(f,e)	f = file variable
	e = error parameter
		Writes the contents of the file buffer into the current component. File f must have relative or indexed organization and be opened for direct or keyed access. The current component must be locked.

(Continued on next page)

Table C-1: (Cont.) Predeclared Procedures

Procedure	Parameter	Action

WRITE(f,p1,...,pn,e)	f = file variable
	p1,...,pn = write parameters
	e = error parameter
		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,e)	f = text file variable
	p1,...,pn = write parameters
	e = error parameter
		Performs the WRITE procedure, then skips to the beginning of the next line. The write parameters are optional. The default for f is OUTPUT.

WRITEV(s,p1,...,pn)	s = character-string variable p1,...,pn = write parameters	Writes the values of p1 through pn into the character string s.

Table C-2: Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
Arithmetic	ABS(x)	Any arithmetic type	Same as x	Computes the absolute value of x.
	ARCTAN(x)	Integer, Unsigned, Real	Real	Computes the arc tangent of x. The result is expressed in radians.
		Double Quadruple	Double Quadruple	
	COS(x)	Integer, Unsigned, Real	Real	Computes the cosine of x. The parameter is expressed in radians.
		Double Quadruple	Double Quadruple	
	EXP(x)	Integer, Unsigned, Real	Real	Computes e**x, the exponential function.
		Double Quadruple	Double Quadruple	
	LN(x)	Integer, Unsigned, Real	Real	Computes the natural logarithm of x. The value of x must be greater than 0.
		Double Quadruple	Double Quadruple	
	SIN(x)	Integer, Unsigned, Real	Real	Computes the sine of x. The parameter is expressed in radians.
		Double Quadruple	Double Quadruple	
	SQR(x)	Any arithmetic type	Same as x	Computes x**2, the square of x.
	SQRT(x)	Integer, Unsigned, Real	Real	Computes the square root of x. If x is less than zero, an error occurs.
		Double Quadruple	Double Quadruple	
Ordinal	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).
Boolean	EOF(f)	File	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	Boolean	Indicates whether the position of file f is at the end of a line. EOLN(f) is TRUE only when the file position is after the last character in a line, in which case the value of f^ is a space. The default for f is INPUT.
	ODD(x)	Integer, Unsigned	Boolean	Returns TRUE if the integer x is odd; returns FALSE if x is even.
	UFB(f)	File	Boolean	Returns TRUE if the last file operation left the file buffer undefined; otherwise, returns FALSE.
	UNDEFINED(r)	Real, Double, Quadruple	Boolean	Returns TRUE if the variable r contains a reserved operand; otherwise, returns FALSE.

(Continued on next page)

Table C-2: (Cont.) Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
Transfer	CHR(x)	Integer, Unsigned	Char	Returns the character (if one exists) whose ordinal value is x.
	DBLE(x)	Any arithmetic type	Double	Rounds the value of x to a double-precision real number.
	INT(x)	Any ordinal type	Integer	Converts the value of x to an integer.
	ORD(x)	Any ordinal type	Integer	Returns the ordinal value corresponding to the value of x.
	QUAD(x)	Any arithmetic type	Quadruple	Rounds the value of x to a quadruple-precision real number.
	ROUND(x)	Real, Double, Quadruple	Integer	Rounds the real value x to the nearest integer.
	SNGL(d)	Any arithmetic type	Real	Rounds the value of d to a single-precision real number.
	TRUNC(x)	Real, Double, Quadruple	Integer	Truncates the real value x to an integer.
	UINT(x)	Any ordinal type	Unsigned	Converts the value of x to an unsigned integer.
	UROUND(r)	Real, Double, Quadruple	Unsigned	Converts the value of a real-type parameter r to an unsigned integer by rounding the fractional part.
	UTRUNC(r)	Real, Double, Quadruple	Unsigned	Converts the value of a real-type parameter r to an unsigned integer by truncating the fractional part.
Dynamic Allocation	ADDRESS(x)	Any VOLATILE variable except a component of a packed structured type	Pointer	Returns a pointer which references the parameter.
Character String	BIN(x,l,d)	x = any type l,d = Integer	Varying	Converts a parameter x to its binary representation. Returns the binary value in a string of length l with d significant digits. Parameters l and d are optional.
	HEX(x,l,d)	x = any type l,d = Integer	Varying	Converts a parameter x to its hexadecimal representation. Returns the hexadecimal value in a string of length l with d significant digits. Parameters l and d are optional.
	INDEX(s1,s2)	Any string type	Integer	Locates the first occurrence of s2 within s1. Returns an integer value indicating the leftmost position of s2. Returns 0 if s2 is not found.
	LENGTH(s)	Any string type	Integer	Returns an integer value indicating the current length of s.
	OCT(x,l,d)	x = any type l,d = Integer	Varying	Converts a parameter x to its octal representation. Returns the octal value in a string of length l with d significant digits. Parameters l and d are optional.

	PAD(s,fill,l)	s = any string type
		fill = Character
		l = Integer
			Varying	Pads a string s with a fill character until it is of length l.

(Continued on next page)

Table C-2: (Cont.) Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
	SUBSTR(s,b,l)	s = any string type b,l = Integer	Varying	Constructs a new string beginning at position b of a given string s and extending to length l.
Unsigned	UAND(u1,u2)	Unsigned	Unsigned	Performs a binary logical AND on the corresponding bits of parameters u1 and u2.
	UNOT(u)	Unsigned	Unsigned	Performs a binary logical NOT on the bits of parameter u.
	UOR(u1,u2)	Unsigned	Unsigned	Performs a binary logical OR on the corresponding bits of parameters u1 and u2.
	UXOR(u1,u2)	Unsigned	Unsigned	Performs a binary logical exclusive OR on the corresponding bits of parameters u1 and u2.
Allocation Size	SIZE(x,c1,...,cn)	x = any type c1,...,cn = case constants	Integer	Returns an integer value indicating the number of bytes allocated for a variable or record field of type x. If the variable is part of a variant record, case constants c1 through cn may be specified.
	NEXT(x)	Any type	Integer	Returns an integer value indicating the number of bytes allocated for a component of type x in an unpacked array.
	BITSIZE(x)	Any type	Integer	Returns an integer value indicating the number of bits allocated for a field of type x in a packed record.
	BITNEXT(x)	Any type	Integer	Returns an integer value indicating the number of bits allocated for a component of type x in a packed array.
Low_Level Interlocked	ADD_INTERLOCKED(e,v)	e = assignment compatible with v v = Integer, Unsigned, or Subrange	Integer	Adds e to v. Returns -1 if the result is negative, 0 if the result is zero, +1 if the result is positive.
	SET_INTERLOCKED(b)	Boolean	Boolean	Assigns TRUE to parameter b and returns its original value.
	CLEAR_INTERLOCKED(b)	Boolean	Boolean	Assigns FALSE to parameter b and returns its original value.
Miscellaneous	CARD(s)	Set	Integer	Returns the number of elements currently belonging to the set s.
	CLOCK	—	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, Quadruple	Integer	Returns the integer-valued exponent of the floating-point representation of r.
	STATUS(f)	File	Integer	Returns 0 if the previous operation on the file succeeded, -1 if the previous operation encountered an end-of-file, and a positive integer representing an error code if the previous operation resulted in an error.

Glossary

actual parameter

Value or variable that is passed in a procedure or function call and is used during execution of the routine.

actual parameter list

List of actual parameters that is specified in a routine call. The actual parameters must be separated by commas, and the entire list enclosed in parentheses.

address

A location in the computer's memory.

arithmetic expression

Expression that uses arithmetic operators to produce a real or integer value.

arithmetic operator

Symbol used with numeric variables, function designators, and constants in forming arithmetic expressions. In PASCAL, the arithmetic operators are +, –, *, /, **, DIV, REM, and MOD.

array

Collection of a specified number 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 VAX–11 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 255, inclusive.

assignment statement

Executable statement that assigns a value to a variable.

base type

(1) The ordinal type of which a subrange is a subset. For example, the subrange 0..123 has the base type INTEGER. (2) The ordinal type from which the members of a 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. The statement executed is the one whose case label is equal to 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, string variable,* and *varying character string.*

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, record, or file, an individual data item. An array component is denoted by the array name and an index for each dimension. A record component (also called a *field*) is denoted by the record name followed by the field name. See also *file component.*

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

constant expression

An expression whose value can be computed when the program is compiled.

control statement

Statement that directs the flow of control in a program, such as the IF-THEN-ELSE, FOR, WHILE, REPEAT-UNTIL, or CASE statement.

control variable

Ordinal variable that takes on sequential values with each iteration of a FOR loop. After normal 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 associates an identifier with what it represents. All labels and all identifiers for constants, types, variables, procedures, and functions must be declared.

declaration section

The part of a block that contains the declarations 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 index of an array. An array can have any number of dimensions; see *multidimensional array.*

double precision

Precision of approximately 16 significant digits and a range from 10**–38 through 10**38, or 15 significant digits and a range from 10**–308 through 10**308 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 constant 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, which perform the actions of the block. See *statement*.

expression

A constant, a variable, a function designator, or some combination of constants, variables, function designators, operators, and parentheses that PASCAL can evaluate. Every expression is associated with a type.

extension

Language feature found in VAX–11 PASCAL, but not in the PASCAL standard proposed by the International Organization for Standardization.

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 value.

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 with a file component.

file name

A 0- to 9-character name component of a VAX/VMS file specification.

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 VAX/VMS 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 VAX/VMS file name, file type, and file version number.

file type

(1) In the VAX/VMS file specification, the 0- to 3-character type component that usually describes the nature of a file or how it is used; for example, PAS indicates a PASCAL source program. (2) In VAX–11 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 with a file component.

floating-point notation

Representation of real-number data in the integer.fraction format, followed by a positive or negative exponent. The exponent is introduced by the letter E for a single-precision number, as in 7.321E02; by the letter D for a double-precision number, as in 9.345D10; and by the letter Q for a quadruple-precision number as in 6.307Q04.

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

Routine that returns a value when executed. A function consists of a heading (which includes the function's name and result type) and a block. See also *predeclared function.*

function designator

Use of a function name and actual parameter list in an executable statement to invoke the function.

function heading

Specification of the name, formal parameter list, and result type of a function in a function declaration.

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.

heading

Specification that precedes a block and defines the block's name and parameters.

identifier

One or more alphanumeric characters that denote a symbolic constant, data type, variable, procedure, function, or other item that is not a reserved word. Although an identifier can be of any length, PASCAL treats only the first 31 characters as significant.

index

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. Input procedures provided by PASCAL include READ, READLN, and GET.

integer

In VAX–11 PASCAL, a whole number between –2,147,483,647 and +2,147,483,647; a value of type INTEGER.

INTEGER type

Predefined scalar type that has the integers as values. In VAX–11 PASCAL, integer values range from –2,147,483,647 through 2,147,483,647.

interactive mode

Mode of communication in which the system responds to the commands and program input that the user types at the terminal.

label

(1) See *case label.* (2) Nonnegative 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 (:).

linker

Program that creates an executable image from one or more object modules. Programs must be linked before they can be executed.

local identifier

Identifier that is declared within a block and is unknown — and therefore inaccessible — outside that block.

logical expression

Expression that uses logical operators to combine Boolean values.

logical operator

Reserved word or symbol that specifies a logical test, the result of which is one of the Boolean values FALSE or TRUE. The logical operators in PASCAL include AND, OR, and NOT.

loop

Construct that controls the repetitive execution of one or more statements until a specified condition is met.

loop body

Statement or group of statements that are executed repetitively under the control of a FOR, REPEAT, or WHILE statement.

modulus

Integer value that results when one operand, J, is repeatedly subtracted from another operand, I (or repeatedly added, if I is negative), until the difference is less than J. This difference is called the modulus of I with respect to J.

multidimensional array

Array with components of an array type. Each dimension of a multidimensional array has its own indexes, which can be of different types.

nested

Contained within, as in a function declared within a procedure or a record that is a field of another record.

nonstandard

Not found in the PASCAL language standard proposed by the International Organization for Standardization; said of extensions that are specific to VAX-11 PASCAL.

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.

operand

Expression whose value is used in an arithmetic, relational, or logical operation.

operator

Symbol used in an expression to cause PASCAL to perform a specific calculation task. PASCAL includes arithmetic, relational, and logical operators.

ordinal type

Sequence of values having a one-to-one correspondence with the set of integers. The ordinal types in VAX-11 PASCAL are INTEGER, UNSIGNED, CHAR, BOOLEAN, enumerated types, and subranges of these ordinal 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, CHAR, 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. Output procedures provided by PASCAL include WRITE, WRITELN, and PUT.

packed

Stored densely in the computer's memory.

parameter

Means of passing information between blocks. See *actual parameter, formal parameter, read parameter,* and *write parameter.*

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) 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 type

Type whose values are references to dynamic variables.

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 an ordinal 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, symbolic constant, file variable, procedure, or function.

predeclared routine

Procedure or function declared by PASCAL and available for use without further declaration.

predefined

See *predeclared.*

procedure

Routine that consists of a procedure heading and a block. When called, a procedure is executed as a unit. See also *predeclared routine.*

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.

quadruple precision

Precision of approximately 33 significant digits and a range from 10**-4932 through 10**4932 for a floating-point real number; the type QUADRUPLE provides quadruple precision.

QUADRUPLE type

Predefined scalar type that has quadruple-precision real numbers as values.

read parameter

Variable used as a parameter in a call to the READ or READLN procedure to which an input value will be assigned.

real number

In VAX–11 PASCAL, the floating-point internal representation of a number ranging from approximately 0.84*(10**–4932) through 0.59*(10**4932) for positive quantities, approximately –0.59*(10**4932) through –0.84*(10**–4932) for negative quantities, and the value 0.0.

REAL type

Predefined scalar type that has the single-precision real numbers as values; synonymous with SINGLE type.

record

Organized collection of data containing zero or more fields, each of which can be of a different type.

relational expression

Expression that uses relational operators to test the relationship between two values.

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 IN.

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 block wherever a *function designator* appears.

routine

Procedure or function; used in this manual in descriptions that apply to both procedures and functions.

scalar 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.

set

Collection of elements of an ordinal type.

single precision

Precision of approximately seven significant digits and a range from 10**-38 through 10**38 for a floating-point real number; the types SINGLE and REAL provide single precision.

SINGLE type

Predefined scalar type that has the single-precision real numbers as values; synonymous with REAL type.

source file

VAX/VMS file that contains source program statements used as input to a language compiler.

statement

Sequence of reserved words, identifiers, operators, expressions, and special symbols that performs a program action or alters 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 type

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).

subrange type

Subset of an existing ordinal type, defined for use as a distinct type. A subrange must be a continuous range of constant values, and is described by its upper and lower bounds separated by the .. symbol.

successor value

Value that succeeds a given value in an ordinal 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 editor

Complex program that allows you to create files and modify existing files.

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 VAX-11 PASCAL — INTEGER, REAL, SINGLE, DOUBLE, QUADRUPLE, BOOLEAN, CHAR, and UNSIGNED. Others can be defined by the programmer.

UNSIGNED type

Predefined scalar type that has an extended set of nonnegative integers as values.

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, symbolic constant, type, variable, procedure, or function.

value initialization

VAX–11 PASCAL extension that allows a programmer to assign a constant value to a variable in the program's declaration section.

value parameter

Formal parameter that represents an actual parameter expression whose value is used as input to a routine.

variable

Data item (of fixed type) that can change in value during execution of the program.

variable parameter

Formal parameter that represents an actual parameter variable whose value can change as a result of the execution of a routine.

varying character string

Sequence of ASCII characters whose values are character strings of different lengths.

write parameter

Expression with optional field width that is specified as a parameter to the WRITE or WRITELN procedure, which writes it in the specified file.

INDEX

A

Addition operator, 2–7
Address, passing, 8–9, 8–10
Apostrophe character, 2–5
  in strings, 6–8
Arithmetic operator, 2–7
ASCII character set, B–1
Assignment, 1–11
  of arrays, 6–4
  of character strings, 6–9
  operator, 1–11, 4–2
  statement, 1–11, 4–1
  to variables, 4–1
Array,
  component, 6–1, 6–2
  data structure, 6–1
  dimension, 6–5
  index, 6–1
  index type, 6–2
  multidimensional, 6–5
  operations on, 6–4
  packed, 6–8
  structured type, 6–1
  three-dimensional, 6–7
  two-dimensional, 6–5
  type definition, 6–2

B

Base type, 3–10
BEGIN delimiter, 1–8, 1–10
BEGIN-END pair, exception, 6–11, 7–7
  use in routines, 8–4
Block, 3–1
  function, 8–7, 8–8
  procedure, 8–6
  program, 1–6
Block structure, 1–1
Boolean expression, 2–10
Boolean type, 2–4
  writing, 5–6
Bounds, of subrange type, 3–7

C

Call, procedure, 8–6
Case,
  label, 7–7
  selector, 7–8
CASE statement, 7–1, 7–7, 7–8
CHAR type, 2–5
Character set, ASCII, B–1
Character string, 2–5, 6–8, 6–9
    See also string
  constant, 6–8
  expression, 6–10
  reading, 6–9, 7–13
  variable, 2–9, 6–8, 6–9
  writing, 6–10
Comment, 1–8
Command,
  EDIT, 1–2, 1–4
  LINK, 1–2, 1–5
  PASCAL, 1–2, 1–4
  PRINT, 1–5
  qualifiers, 1–3
  RUN, 1–2, 1–5
Compiling a program, 1–2, 1–4
Component,
  array, 6–1
  file, 5–1
Compound statement, 1–11, 4–3
Conditional statement, 1–11, 2–9
  CASE, 7–7
  IF-THEN, 4–3
  IF-THEN-ELSE, 4–5

CONST section, 3-1, 3-4
Constant,
  definition, 3-4
  expression, 3-4
  string, 6-8
  symbolic, 3-4
  type of, 2-1
Control statement, 1-11, 4-1
Control variable, 4-8
Creating a program, 1-4
CTRL/Z, specifying EOF with, 5-12

D

Data,
  global, 8-5
  local, 8-5
  reading, 5-2
  structures, 6-1
  type, 2-1
  writing, 5-5
Decimal format, printing, 5-7
Decimal notation, 2-3
Declaration,
    See also definition
  FUNCTION, 3-1, 8-7
  LABEL, 3-1
  order of, 3-1
  PROCEDURE, 3-1, 8-6
  section, 1-6, 1-9, 3-1
  VAR, 1-9, 3-6
  variable, 2-5, 3-6
Default,
  values for field width, 5-6
  values for file specification, 1-3
Defining types, 3-5
  in TYPE section, 3-8
  in VAR section, 3-8
Definition, 3-1
    See also declaration
  array, 6-1
  CONST, 3-1, 3-4
  enumerated type, 3-8
  file, 6-1
  record, 6-1, 6-10
  set, 6-1
  subrange type, 3-10
  TYPE, 1-9, 3-1, 3-5, 3-8
  type in VAR section, 3-8
  varying character string, 6-1
Delimiter,
  BEGIN, 1-8, 1-10, 4-3
  END, 1-8, 1-10, 4-1, 4-3
  period, 1-8
  semicolon, 1-8, 4-1
Dimension, of array, 6-5
DIV operator, 2-7, 2-8
Division operator, 2-7, 2-8
DOUBLE type, 2-4
  writing, 5-6, 5-7, 5-8
Double-precision, 2-2
DOWNTO, 4-8
Dynamic variable, 2-2

E

EDIT command, 1-2, 1-4
Editor,
  EDT, 1-4
End-of-file,
  condition, 1-11, 5-11
  condition, on a terminal, 5-12
End-of-line,
  condition, on a terminal, 5-11
END,
  delimiter, 1-8, 1-10, 4-1, 4-3
  in record type definition, 6-11
  in CASE statement, 7-7
Enumerated type,
  definition, 3-8
  writing, 5-6, 5-7, 5-9
EOF function, 1-11, 5-10, 5-11
EOLN function, 5-10
Error message, 1-4, 1-5, 1-6
Executable,
  image, 1-2, 1-5
  section, 1-6, 1-10
  statement, 1-6, 1-10
Executing a program, 1-2, 1-5
Exponential notation,
    see floating-point notation
Exponentiation operator, 2-7, 2-8
Expression, 1-11, 2-6
  arithmetic, 2-7, 2-8
  Boolean, 2-9
  character string, 2-9, 6-10
  constant, 3-4
  evaluation of, 2-11
  logical, 2-9, 2-10
  operator in, 2-6
  parentheses in, 2-11, 2-12
  relational, 2-9
  type of, 2-1
  writing, 5-5
External file, 1-6

F

Field,
  of a record, 6-11
  width, 1-10, 5-6
  width rules, 5-7

File,
access to, 5–2
component, 5–1
data structure, 6–1
external, 1–6
INPUT, 1–9, 5–1
name, 1–3
OUTPUT, 1–9, 5–1
position, 5–2
position, after READ, 5–3, 5–12
position, after READLN, 5–4, 5–12
position, after WRITE, 5–6
position, after WRITELN, 5–9
position, at EOF, 5–11
position, at EOLN, 5–11
predeclared text, 1–9
source, 1–4
specification, 1–3
specification defaults, 1–3
type, VAX/VMS, 1–3
variable, 5–1
FILE type, 5–1, 6–1
Floating-point notation, 2–4
writing numbers in, 5–7, 5–8
Flow chart,
CASE statement, 7–9
FOR statement, 4–9
IF-THEN statement, 4–4
IF-THEN-ELSE, 4–6
REPEAT statement, 7–2
WHILE statement, 7–5
FOR statement, 4–8
Format, of program, 1–8
Function, 8–1, 8–7
block, 8–8
declaration, 8–1, 8–7
designator, 8–1, 8–8
heading, 8–7
predeclared, C–5
result type of, 2–1, 8–7
FUNCTION section, 3–1, 8–7

G

Global, identifier, 8–5
GOTO statement, 7–1

H

Heading,
function, 8–7
procedure, 8–6
program, 1–6, 1–8
routine, 8–5

I

Identifier,
constant, 3–4
in enumerated type, 3–8
global, 8–5
local, 8–5
predeclared, 3–2, A–2
redeclaring, 8–5
scope of, 8–5
user, 3–3
IF-THEN statement, 4–3
IF-THEN-ELSE statement, 4–5
Index,
in multidimensional array, 6–5, 6–6, 6–7
type, 6–1, 6–2, 6–3, 6–4
Information-level,
diagnostic messages, 1–4
errors, 1–4
Initializing a variable, 1–9, 3–6
Input procedure, 1–10, 5–1, 5–2
INPUT file, 1–9, 5–1
Integer, writing, 5–6, 5–7
INTEGER type, 2–3
Interactive program, 5–1

L

LABEL section, 3–1
LINK command, 1–2, 1–5
Linking an object module, 1–2, 1–5
/LIST qualifier, 1–5
Logical operator, 2–9
Local data, 8–5
Local identifier, 8–5
Loop, 4–8, 7–1
use with arrays, 6–6
Loop body, 4–8, 7–1
Lowercase characters in identifier, 3–3

M

MOD operator, 2–7, 2–8
Module, object, 1–2, 1–4
Modular programming, 1–1
Multidimensional array, 6–2, 6–5
Multiplication operator, 2–7, 2–8

N

Nonstandard features,
suppressing error messages for, 1–5
/NOSTANDARD qualifier, 1–5, 1–6

O

Object module, 1–2, 1–4
  linking, 1–5
Operator,
  addition, 2–7
  DIV, 2–7, 2–8
  division, 2–7, 2–8
  exponentiation, 2–7, 2–8
  logical, 2–9
  MOD, 2–7, 2–8
  multiplication, 2–7, 2–8
  precedence, 2–11
  relational, 2–9, 6–10
  REM, 2–7, 2–8
  subtraction, 2–7, 2–8
ORD function, 2–3, 2–4, 2–5, 3–8
Ordering,
  of declaration section, 3–1
  of enumerated types, 3–8
Ordinal type, 2–3
  BOOLEAN 2–4
  CHAR, 2–5
  enumerated, 3–8
  INTEGER, 2–3
  subrange, 3–10
Ordinal value, 2–3
  of enumerated types, 3–8
  of subrange types, 3–10
OTHERWISE clause,
  in CASE statement, 7–8
Output procedure, 1–10, 5–1, 5–5
OUTPUT file, 1–9, 5–1

P

Packed, 6–8
Parentheses in expressions, 2–11, 2–12
Parameter, 8–5, 8–8
  actual, 8–5, 8–6, 8–8, 8–9
  actual list, 8–6, 8–9
  correspondence of, 8–9
  declaration, 8–5, 8–9
  formal, 8–5, 8–6, 8–8, 8–9
  formal list, 8–5, 8–9
  list, 8–5, 8–8, 8–9
  read, 5–2
  type of, 8–9
  value, 8–10, 8–11, 8–12
  VAR, 8–9, 8–10, 8–11
  variable, 8–9, 8–10, 8–11, 8–12
  write, 5–5
PASCAL,
  command, 1–2, 1–4
  defined names, A–1
  VAX–11 extensions, 1–2, 1–4
Pass data to routine, 8–8
Pass by reference,
    See VAR parameter
Pass by value,
    See value parameter
Period delimiter, 1–8, 1–10
Pointer type, 2–1, 2–2
Predecessor value, 4–9
Predeclared,
  function, C–5
  identifier, 3–2, A–2
  procedure, C–1
  routine, 8–1, C–1
Precedence rules, 2–11
PRINT command, 1–5
Procedure, 8–1, 8–6
  block, 8–6
  call, 8–6
  declaration, 8–6
  heading, 8–6
  input, 1–10, 5–2
  name, 8–6
  output, 1–10, 5–5
  predeclared, C–1
PROCEDURE section, 3–1
Program,
  block, 1–6
  compiling, 1–4
  development, 1–2, 1–3
  execution, 1–2, 1–5
  heading, 1–6, 5–1, 5–2
  interactive, 5–1
  listing, creating, 1–5
  structure of, 1–6
Prompting at the terminal, 7–13

Q

QUADRUPLE type, 2–2
  writing, 5–6, 5–7, 5–8
Quadruple precision, 2–2
Qualifier,
  on command, 1–3
  /LIST, 1–5
  /NOSTANDARD, 1–5, 1–6

R

Read,
  of character data, 5–3
  of character string, 6–9, 7–13
  of data, 5–1, 5–2
  parameter, 5–2

READ procedure, 5–2
READLN procedure, 5–2, 5–4
Real number,
  writing, 5–6, 5–7, 5–8
REAL type,
  double-precision, 2–2
  single-precision, 2–2
  quadruple-precision, 2–2
Relational operator, 2–9, 6–10
Record,
  data structure, 6–1
  field of, 6–11
  nested, 6–12
  structured type, 6–1
  type definition, 6–11
REPEAT statement, 7–1
Repetitive statement, 2–9
  REPEAT, 7–1
  WHILE, 7–5
Reserved words, 3–2, A–1
Result type, 2–1, 8–7
Routine, 8–1
  block, 8–5
  declaration, 8–4
  execution, 8–4
  heading, 8–5
  nested, 8–4
  passing data to, 8–8
RUN command, 1–2, 1–5

S

Scalar type, 2–1
  BOOLEAN, 2–4
  CHAR, 2–5
  enumerated, 3–8
  INTEGER, 2–3
  ordering of, 2–3
  REAL, 2–3
  subrange, 3–10
Scientific notation,
    See floating-point notation
Section,
  CONST, 3–3
  declaration, 1–6, 3–1
  executable, 1–6, 1–10
  FUNCTION, 3–1, 8–7
  LABEL, 3–1
  PROCEDURE, 3–1, 8–6
  TYPE, 3–1, 3–5, 3–7
  VAR, 3–1, 3–5, 3–8
Semicolon delimiter, 1–8, 4–1, 8–6, 8–8
SET type, 6–1
Single-precision, 2–2
SINGLE type, 2–2
  writing, 5–6, 5–7, 5–8
Source file, 1–2, 1–4
Statement, 4–1
  assignment, 1–11, 4–1
  CASE, 7–1, 7–7
  compound, 1–11, 4–1, 4–3
  conditional, 1–11, 2–9, 4–3, 4–5
  control, 1–11, 4–1
  executable, 1–6, 1–10
  FOR, 4–8
  GOTO, 7–1
  IF-THEN, 4–3
  IF-THEN-ELSE, 4–5
  procedure call, 8–6
  REPEAT, 1–11, 7–1
  repetitive, 2–9
  WHILE, 7–1, 7–5
  WITH, 6–13
String,
  character, 6–8
  comparing, 6–10
  constant, 6–8
  expression, 6–10
  input of, 6–9
  ordering of characters in, 6–10
  operation, 6–10
  output of, 6–10
  read with READ or READLN, 6–9
  print with WRITE or WRITELN, 6–10
  variable, 2–9, 6–8, 6–9
Structured type, 2–1
  array, 6–1
  record, 6–11
  variable, 6–1
Subrange type, 3–10
Subscript,
    See index
Subtraction operator, 2–7, 2–8
Successor value, 4–9
Symbolic names, 3–1
Syntax rules, 1–8
  for user identifiers, 3–3

T

Text file, 5–1, 5–10
  end of, 5–11
  predeclared, 1–9, 5–1
Terminal,
  prompting at, 7–13
  input from, 5–1
  output to, 5–1

Three-dimensional array, 6–7
TO, 4–8
Two-dimensional array, 6–5
Type, 2–1
  ARRAY, 6–1
  BOOLEAN, 2–4
  CHAR, 2–5
  of constant, 2–1
  DOUBLE, 2–2, 2–4
  of expression, 2–1
  file, 1–3
  FILE, 5–1, 6–1
  of function, 2–1, 8–7
  function result, 8–8
  identifier, 3–5
  INTEGER, 2–3
  ordinal, 2–3
  pointer, 2–1, 2–2
  predefined, 2–2
  QUADRUPLE, 2–2, 2–4
  REAL, 2–3
  RECORD, 6–11
  SET, 6–1
  SINGLE, 2–2
  subrange, 3–7, 3–10
  UNSIGNED, 2–2
  user-defined scalar, 3–7
  of variable, 2–1
  VARYING, 6–1
TYPE section, 3–1, 3–5, 3–8
Type definition, 1–9, 3–5
  array, 6–2
  enumerated, 3–8
  record, 6–11
  subrange, 3–10

U

UNSIGNED type, 2–2
UNTIL clause,
  of REPEAT statement, 1–11, 7–4
Uppercase characters in identifier, 3–3
User-defined,
  enumerated type, 3–8
  scalar type, 3–7
  subrange type, 3–10
  types, 3–5, 3–7, 3–8
User identifiers, 3–3
  dollar sign in, 3–3
  syntax rules for, 3–3

V

Value parameter, 8–10, 8–11, 8–12
Value initialization, 3–6
  illegal use of, 8–6, 8–8
VAR,
  declaration, 1–9
  parameter, 8–9, 8–10, 8–11
  section, 3–1, 3–6, 3–7
Variable, 1–9, 1–11, 2–5
  assigning values to, 4–1
  control, 4–8
  declaration, 2–5, 3–6
  declaration, format of, 3–6
  dynamic, 2–2
  file, 5–1
  identifier, 3–6
  initializing, 1–9, 3–6
  record, 6–11
  section,
    See VAR section
  string, 2–9, 6–8, 6–9
  structured type, 6–1
  type of, 2–1
Variable parameter, 8–9, 8–10, 8–11, 8–12
Varying character string, 6–1

W

Warning-level,
  diagnostic messages, 1–4
  errors, 1–4
WHILE statement, 7–1, 7–5
WITH statement, 6–13
Write,
  of character string, 5–5, 5–6, 5–7, 6–10
  of data, 5–5
  of expressions, 5–5
  parameter, 5–5
WRITE procedure, 5–5
WRITELN procedure, 5–5, 5–9

VAX-11 PASCAL Primer
AA-J180B-TE

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 user/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 ____________

Street ____________

City ____________ State ____________ Zip Code or Country ____________

BC 25714

— — — 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

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

— — — Do Not Tear - Fold Here —