March 1981

This document describes the elements of the Pascal language supported by VAX-11 PASCAL. It is intended as a reference manual for use in preparing VAX-11 PASCAL source programs.

VAX-11 PASCAL
Language Reference Manual

Order No. AA-H484B-TE

SUPERSESSION/UPDATE INFORMATION: This revised document supersedes the VAX-11 PASCAL Language Reference Manual (Order No. AA-H484A-TE)

SOFTWARE VERSION: VAX-11 PASCAL V1.2

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, November 1979
Revised, March 1981

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

The following are trademarks of Digital Equipment Corporation:

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

ZKA23-81

CONTENTS

CONTENTS

CONTENTS

CONTENTS

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual describes the VAX-11 PASCAL language. The manual is designed primarily for reference; it is not a tutorial document. For information about tutorial and user documents, refer to the list below under "Associated Documents."

INTENDED AUDIENCE

This manual is intended for readers who know the PASCAL language. The reader need not have a detailed understanding of the VAX/VMS operating system, but some familiarity with VAX/VMS is helpful. For information about VAX/VMS, refer to the documents listed below under "Associated Documents."

STRUCTURE OF THIS DOCUMENT

This manual contains seven chapters and six appendixes.

- Chapter 1 describes the format of a PASCAL program, showing its structure and elements.
- Chapter 2 introduces basic concepts including constants, variables, data types, expressions, and scope of identifiers.
- Chapter 3 describes the program heading and declaration section.
- Chapter 4 provides detailed information on PASCAL data types.
- Chapter 5 describes the statements that perform the actions of a program.
- Chapter 6 explains the use of functions and procedures, and summarizes the predeclared functions and procedures supplied with VAX-11 PASCAL.
- Chapter 7 provides detailed information on input and output procedures.
- Appendix A lists the ASCII character set.
- Appendix B presents the VAX-11 PASCAL language in the Backus-Naur form and includes syntax diagrams.
- Appendix C summarizes the extensions incorporated in VAX-11 PASCAL.

- Appendix D describes how to specify compiler qualifiers in the source code.
- Appendix E contains complete PASCAL program examples.
- Appendix F describes the Version 1.0 OPEN procedure and the difference between it and Version 1.2.

ASSOCIATED DOCUMENTS

Users at all levels should refer to the VAX-11 PASCAL User's Guide for information on compiling, linking, running, and debugging their programs.

For programmers unfamiliar with the PASCAL language, the VAX-11 PASCAL Primer provides a tutorial introduction.

The VAX/VMS Primer provides introductory material for programmers unfamiliar with the VAX/VMS operating system.

The VAX/VMS Command Language User's Guide describes the VAX/VMS commands and will help all users in creating, editing, copying, and printing files containing PASCAL programs.

The VAX-11 Information Directory and Index briefly describes all system documentation, defining the intended audience for each manual and providing a synopsis of each manual's contents.

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: {expr {statement}
...	A horizontal ellipsis means that the preceding item can be repeated as indicated, for example: filename, ...
. . .	A vertical ellipsis means that not all of the statements in a figure or example are shown.
[[ ]]	Double brackets in the statement format descriptions enclose items that are optional, for example: [[PACKED]] Double brackets in statement and declaration format description enclose items that are optional, for example: WRITE ([[OUTPUT,]] print list)

Convention	Meaning
[ ]	Square brackets mean that the statement syntax requires the square bracket characters. This notation is used with arrays and sets, for example: ARRAY [subscriptl]
items in uppercase letters and special symbols	Uppercase letters and special symbols in format descriptions indicate PASCAL reserved words that you must not abbreviate, for example: BEGIN END
items in lowercase letters	Lowercase letters represent elements that you must replace according to the description in the text.

SUMMARY OF TECHNICAL CHANGES

This section summarizes the technical changes in the use of the VAX-11 PASCAL language for Version 1.2.

The OPEN procedure now allows the use of a string variable for the file name and for nonpositional parameters.

The length of user-defined identifiers has been expanded from 15 to 31 characters.

CHAPTER 1

INTRODUCTION

VAX-11 PASCAL is an extended implementation of the PASCAL language. Developed for use under the VAX/VMS operating system, PASCAL includes all the standard language elements plus the following extensions:

- Exponentiation operator
- Hexadecimal and octal integers
- Double-precision real data type
- Dollar sign ($) and underscore (_) characters in identifiers
- External procedure and function declarations
- CARD, CLOCK, EXPO, SNGL, and UNDEFINED functions
- UPPER and LOWER bound functions for arrays
- OTHERWISE clause in the CASE statement
- OPEN and CLOSE procedures for file access
- FIND procedure for direct access to sequential files
- Optional carriage control for output files
- DATE, TIME, HALT, and LINELIMIT procedures
- VALUE initialization section
- Dynamic array parameters
- Separate compilation
- %INCLUDE directive for alternate input files during compilation
- Extended parameter specifications to support the VAX-11 procedure calling standard

1.1 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 the external files the program uses. The block is divided into two parts: the declaration section, which contains data declarations, and the executable section, which contains executable statements. Figure 1-1 points out each part of a sample PASCAL program.

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

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

    PROCEDURE Instructions;                            } Procedure Heading

Procedure Block
        BEGIN
        WRITELN ('This program adds, subtracts, multipl : , and');
        WRITELN ('divides real numbers. Enter a number i   esponse');
        WRITELN ('to the Operand: prompt and enter an op  for --- ');
        WRITELN ('+, -, *, /, or = -- in response to the   rator:');
        WRITELN ('prompt. The program keeps a running subt  al');
        WRITELN ('until you enter an equal sign (=) in res  se to');
        WRITELN ('the Operator: prompt. You can then exit f  m');
        WRITELN ('the program or begin a new set of calcula i ns.')
        END; (*end of procedure Instructions*)

Executable Section
    BEGIN

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

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

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

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

ZK-026-80

Figure 1-1 Structure of a PASCAL Program

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

1.1.1 The Program Heading

The program heading begins a PASCAL program and consists of the word PROGRAM followed by the program's name. After the name may be a list of file identifiers, in parentheses. The file identifiers specify the external files the program uses to read data and to record results.

1.1.2 The Declaration Section

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

The declaration section contains the following declarations and definitions, which must appear in the order listed:

1. Labels
2. Constants
3. Types
4. Variables
5. Value initializations
6. Procedures and functions

1.1.3 The Executable Section

The final part of a PASCAL program is the executable section and contains the statements that specify the actions of the program.

The executable section is delimited by the reserved words BEGIN and END. Between BEGIN and END are statements that read, write, and change the values of data items, along with other statements that control execution.

1.2 CHARACTER SET

VAX-11 PASCAL uses the full American Standard Code for Information Interchange (ASCII) character set (see Appendix A). The ASCII character set contains 128 characters in the following categories:

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

The VAX-11 PASCAL compiler does not distinguish between uppercase and lowercase characters except in character and string constants and the values of character and string variables. For example, the reserved word PROGRAM has the same meaning when written as any of the following:

```
PROGRAM

PRogrAm

program
```

The constants below, however, represent different characters:

```
'b'

'B'
```

The following two constants represent different strings:

```
'BREAD AND ROSES'

'Bread and Roses'
```

1.3 STATEMENTS

The basic unit of a PASCAL program is the statement, which directs the system to perform a specified task.

Statements are composed of reserved words, identifiers, and special symbols, combined with user-supplied values.

1.3.1 Reserved Words

VAX-11 PASCAL reserves the words in Table 1-1 as names for statements, data types, and operators. This manual shows these words in uppercase characters.

Table 1-1
Reserved Words

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

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

1.3.2 Identifiers

PASCAL uses identifiers to name programs, modules, constants, types, variables, procedures, and functions. An identifier is a sequence of letters, digits, dollar signs ($), and underscores (_), with the following restrictions:

- An identifier must start with a letter.
- An identifier must be unique in the first 31 characters within the block in which it is declared.
- An identifier must not contain any blanks.

VAX-11 PASCAL places no restrictions on the length of identifiers, but scans only the first 31 characters for uniqueness. The following are examples of valid and invalid identifiers:

Valid

```
FOR2N8
MAX_WORDS
UPTO
LOGICAL_NAME_TABLE          (unique in first
LOGICAL_NAME_SCANNER         31 characters)
SYS$CREMBX
```

Invalid

```
4AWHILE (starts with a digit)
_BAR (starts with an underscore)
UP&TO (contains an ampersand)
YEAR_END_80_MASTER_FILE_TOTAL_DISCOUNT     (not unique in first
YEAR_END_80_MASTER_FILE_TOTAL_DOLLARS       31 characters)
```

Although VAX-11 PASCAL allows the dollar sign ($) in identifiers, this character has a special meaning to the VAX/VMS operating system in some contexts. You should restrict the use of the dollar sign ($) to identifiers representing VAX/VMS symbolic names.

1.3.2.1 **Predeclared Identifiers** - VAX-11 PASCAL reserves the following predeclared identifiers in Table 1-2 as names of functions, procedures, types, values, and files. These predeclared identifiers appear in uppercase characters throughout this manual.

Table 1-2
Predeclared Identifiers

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

You can redefine a predeclared identifier to denote some other item. Doing so, however, means that you can no longer use the identifier for its usual purpose within the scope of the block in which it is redefined (see Section 2.6 for a description of scope of identifiers).

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

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

1.3.2.2 **User Identifiers** - User identifiers denote program and module names, constants, variables, procedures, functions, and user-defined types. User identifiers represent significant data structures, values, and actions that are not represented by a reserved word, predeclared identifier, or special symbol.

1.3.3 Special Symbols

Special symbols represent delimiters, arithmetic, relational, and set operators, and other syntax elements. VAX-11 PASCAL includes the special symbols listed in Table 1-3.

Table 1-3
Special Symbols

Name	Symbol	Name	Symbol
Addition	+	Less than or equal	<=
Assignment operator	:=	Left parenthesis	(
Brackets	[]	Multiplication	*
Colon	:	Not equal	<>
Comma	,	Percent	%
Comments	(* *) { }	Period	.
Division	/	Pointer	^
Equal	=	Right bracket	]
Exponentiation	**	Right parenthesis	)
Greater than	>	Semicolon	;
Greater than or equal	>=	Subrange operator	..
Left bracket	[	Subtraction	-
Less than	<		

1.4 DELIMITERS

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

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

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

You could also put the entire program on one line:

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

As long as each complete statement is separated from the next by a semicolon, PASCAL interprets your input correctly. Spaces, tabs, and carriage-return/line-feed combinations, however, make your program easier to read and understand. Programming examples in this manual are written with one statement on each line. For example:

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

The reserved words BEGIN and END are also used as delimiters. BEGIN indicates the start of the executable section or a compound statement (see Section 5.1), and need not be followed by a semicolon.

END indicates the end of one of the following:

- A record definition (see Section 4.5)
- An executable section
- A compound statement
- A CASE statement (see Section 5.3.1)

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

1.5 DOCUMENTING YOUR PROGRAM

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

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

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

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

```
(*This is a comment, too.*)
```

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

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

1.6 THE %INCLUDE DIRECTIVE

The %INCLUDE directive allows you to read statements from one PASCAL file during compilation of another. The contents of the included file are inserted at the point where the VAX-11 PASCAL compiler encounters the %INCLUDE directive. The %INCLUDE directive is described in this chapter because it has no effect on program execution except to direct the compiler to read PASCAL source from a second file. The %INCLUDE directive can appear anywhere that a comment is legal. %INCLUDE is useful when the same information is used by several programs.

Format

```
%INCLUDE 'VAX/VMS file-specification      {/LIST  }
                                          {/NOLIST}
```

VAX/VMS file-specification

Designates the file to be included (see the VAX-11 PASCAL User's Guide for the form of the VAX/VMS file specification). Apostrophes are required to enclose the VAX/VMS file-specification and /LIST or /NOLIST option.

/LIST

Indicates the included file is to be printed in the listing. LIST is the default.

/NOLIST

Indicates the included file is not printed in the listing.

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

An included file can contain any PASCAL declarations or statements. However, the declarations in an included file, when combined with the other declarations in the compilation, must follow the required order of declarations.

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

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

Main PASCAL Program

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

   CONST %INCLUDE 'CONDEF.PAS/LIST'

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

CONDEF.PAS

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

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

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

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

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

CHAPTER 2

PASCAL CONCEPTS

This chapter introduces general concepts of VAX-11 PASCAL, describing the following aspects of PASCAL programming:

- Numbers
- Constants
- Variables
- Types
- Expressions
- Scope of identifiers

2.1 NUMBERS

VAX-11 PASCAL recognizes integers and real numbers. An integer is entered in decimal, octal, or hexadecimal form. A real number can be specified as single or double precision. The sections below describe the format and acceptable range of values for integers and real numbers.

2.1.1 Integers

Integers are positive and negative whole numbers ranging from -2**31 to 2 ** 31 - 1. This range contains numbers from -2,147,483,648 through 2,147,483,647. The following are valid integers in PASCAL:

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

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

In addition to decimal integers, VAX-11 PASCAL allows you to specify integers in octal and hexadecimal notation. To specify an octal integer, place a percent sign (%) and the letter O in front of the octal number. The letter O can be uppercase or lowercase. For example:

```
%O7712
%O6
%o473
%O150
```

To specify a hexadecimal integer, place a percent sign (%) and the letter X in front of the hexadecimal number. The letter X can be uppercase or lowercase. For example:

```
%X53A1
%XDEC
%x99
%X2C12
```

You can use an octal or hexadecimal integer anywhere an integer is used.

2.1.2 **Real Numbers**

Real numbers are decimal numbers ranging from approximately 0.29*(10**(-38)) to 1.7*(10**38) for positive quantities and -0.29*(10**(-38)) to -1.7*(10**38) for negative quantities.

In a PASCAL program, real numbers are written in decimal notation or scientific notation. The following numbers are in decimal notation:

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

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

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

```
  2.3E2
 -0.07e4
  10.0E-1
-201E+3
  -2.14159E0
```

The letter E after the value means that the value is to be multiplied by a power of 10. Note that you can use an uppercase or lowercase E. The integer following the E tells which power of 10 and is positive or negative. The real number 237.0 is written in any of the following ways:

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

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

VAX-11 PASCAL provides single and double precision for real numbers. Single precision typically provides seven significant digits. Double precision extends the number of significant digits to 16.

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

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

The integer following the D is an exponent, as in single-precision floating-point numbers. All the above values have approximately 16 significant digits.

2.2 CONSTANTS

A constant is a quantity with an unchanging value. The value can be any of the following:

- An integer, such as 7, -2, +3001, %XDEFACE
- A real number, such as 3.1415927, -7.0, 0.631E+5, 19D-4
- A character, such as 'A', '?'
- A string of characters, such as '********', 'out to lunch'
- One of the Boolean values: TRUE or FALSE
- A value of an enumerated type, such as BLUE, MONDAY

Numeric values must be specified as shown in Sections 2.1.1 and 2.1.2. You can use decimal, octal, and hexadecimal integers and single- and double-precision real numbers.

You must enclose character and string constants in apostrophes, for example:

```
'b'

'Blaise'

'Many minds are not sound.'    (*Blaise Pascal*)

'Thought constitutes man''s greatness'    (*Blaise Pascal*)
```

A character string can contain any ASCII character. To use the apostrophe in a string, type it twice as in the fourth string above. A string constant can occupy only one line.

Values of Boolean and enumerated types (see Section 2.4) can also be used as constants. To specify one of these values, use its constant identifier.

You can define an identifier to name a constant value, and then use the identifier anywhere in the program in place of the value (see Section 3.3). For example, if the identifier PI represents the value 3.1415927, you can simply specify PI whenever the value 3.1415927 is needed. You cannot change a constant's value after you define it.

VAX-11 PASCAL includes the predefined constant identifier MAXINT. MAXINT represents 2,147,483,647, which is the largest integer value you can use in a PASCAL program.

2.3 VARIABLES

A variable is a quantity whose value can change while the program executes. In PASCAL, every variable has an identifier, a type, and a value, and must be declared in the declaration section. The identifier and type are permanent characteristics; you cannot change them, except by changing the variable declaration.

A variable's type automatically establishes three other properties:

- The range of values the variable can assume
- The legal operations for the variable
- The predeclared procedures and functions that apply to the variable

The type implicitly indicates how much storage space is required for all the possible values the variable can assume. A variable can change in value any number of times, but all its values must be within the range established by its type.

A variable does not assume a value until assigned one by the program. You can use an assignment statement or an input procedure to assign a value to a variable at execution time. VAX-11 PASCAL also provides the VALUE section (see Section 3.6), which allows you to initialize variables in the declaration section.

2.4 TYPES

In PASCAL, data types are divided into three categories: scalar, structured, and pointer types. The scalar types, introduced in Section 2.4.1, are the fundamental types. They serve as building

blocks for the structured types, introduced in Section 2.4.2. PASCAL also includes a dynamic type called the pointer type, introduced in Section 2.4.3. When you form an expression, assign a value to a variable, or pass parameters to a subprogram, the types involved must follow the rules for type compatibility. Section 2.4.4 introduces the concept of type compatibility.

2.4.1 Scalar Types

A scalar type is an ordered group of values. For example, the scalar type INTEGER denotes the positive and negative whole numbers. The integers follow a certain order, for example, -700 is less than 2.

PASCAL defines some scalar types and allows you to define others to fit your needs. A user-defined scalar type can be defined by enumerating each value or it can be defined as a subrange of another scalar type. Values of user-defined scalar types, like those of predefined scalar types, follow a particular order. The ORD function (see Section 2.4.1.3) returns the ordinal value of a scalar data item.

2.4.1.1 **Predefined Scalar Types** - PASCAL defines the scalar types INTEGER, REAL, CHAR, and BOOLEAN for integer, real number, character, and Boolean values. Two additional predefined types, SINGLE and DOUBLE, provide explicit single- and double-precision real numbers. (Throughout this manual, the term "real types" refers to the REAL, SINGLE, and DOUBLE types.) Sections 2.1.1 and 2.1.2 describe the range of values for variables of the integer and real types.

A value of type CHAR is a single element of the ASCII character set, as listed in Appendix A. To specify a character, enclose it in apostrophes. To indicate the apostrophe character, type it twice within apostrophes. Each of the following is a valid character value:

```
'A'
'z'
'0'
'.'
'''' (the apostrophe)
'?'
```

Variables of type CHAR are always single characters. You can use strings such as 'HELLO' and '***', but they must be represented as packed arrays of characters (see Section 2.4.2.1).

The BOOLEAN type has two values: TRUE and FALSE. PASCAL defines them as predeclared identifiers and orders them so that FALSE is less than TRUE. Boolean values are the result of testing expressions for truth or validity. The result of a relational expression (for example, A<B) is a Boolean value.

2.4.1.2 **User-Defined Scalar Types** - PASCAL not only provides predefined scalar types, but also allows you to define your own scalar types. A user-defined scalar type can be an enumerated type or a subrange of any scalar type except the real numbers.

An enumerated type consists of an ordered list of identifiers. The identifiers are the constant values of the type you are defining. For example, you could define the enumerated type Weekdays with values Monday, Tuesday, Wednesday, Thursday, and Friday.

A subrange type consists of a continuous range of values of another scalar type, called the base type. You can use a subrange anywhere you can use its base type. The subrange symbol (..) is used in specifying a subrange, as follows:

```
1..30
```

This notation specifies a subrange containing the integers from 1 to 30, inclusive. You can also define subranges of character and enumerated types. For example:

```
'A'..'Z'

Monday..Wednesday
```

The subrange in the first example above contains all the ASCII characters from uppercase A to uppercase Z. Assuming you have defined the type Weekdays as previously mentioned, the second example specifies a subrange with the values Monday, Tuesday, and Wednesday.

2.4.1.3 **The ORD () Function** - Each element of a scalar type (except the real types) has a unique ordinal value, which indicates its position in a list of elements of its type. The ORD() function returns the ordinal value as an integer. For example:

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

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

```
ORD('0') is less than ORD('9') and
ORD('A') is less than ORD('Z'), but
ORD('Z') is less than ORD('a')
```

You can use ORD () on a value of an enumerated type, as follows:

```
ORD(Tuesday)
```

Assuming that Tuesday is a value of type Weekdays (which has constants Monday, Tuesday, Wednesday, Thursday, and Friday), this expression returns the integer 1. Enumerated types are ordered starting at 0.

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

2.4.2 Structured Types

PASCAL has four structured types: arrays, records, files, and sets. Using structured types, you can process groups of scalar, structured, or pointer data items. For example, you can have an array of integers, an array of arrays, a record of integers and characters, a file of records, or a set of an enumerated type.

PASCAL allows you to pack variables of structured types to save storage space. Packed structures are stored as densely as possible. The VAX-11 PASCAL User's Guide describes storage allocation for packed and unpacked variables.

2.4.2.1 **Arrays** - An array is a group of variables of the same type that share an identifier. Each variable in the array is called an element of that array, and is referred to with the array identifier and one or more subscripts (or indexes). The subscripts need not be integers; they can be values of any scalar type except a real type.

For example, you could declare an array variable Bat_Avg, with subscripts 1 to 9 and elements that specify the batting average for each player in the starting line-up of a baseball team. To refer to the elements of this array, you would specify Bat_Avg[1], Bat_Avg[2], and so on up to Bat_Avg[9].

Multidimensional Arrays

Arrays with more than one subscript are multidimensional arrays. These are simply arrays of arrays. You can specify up to 255 dimensions and the subscripts need not be of the same type. For example, you could create a 2-dimensional array of batting averages indexed by the name of the team and an integer from 1 to 9. Given the enumerated type League with team name values Stingers, Big_Red, Wolves, and Lizards, the references to this array would be the following:

```
Bat_Avg[Stingers,1]     Bat_Avg[Stingers,2]  ...  Bat_Avg[Stingers,9]
Bat_Avg[Big_Red,1]      Bat_Avg[Big_Red,2]   ...  Bat_Avg[Big_Red,9]
Bat_Avg[Wolves,1]       Bat_Avg[Wolves,2]    ...  Bat_Avg[Wolves,9]
Bat_Avg[Lizards,1]      Bat_Avg[Lizards,2]   ...  Bat_Avg[Lizards,9]
```

Each element of this array contains the batting average of the nth player in the starting line-up of the specified team.

Character String Variables

In PASCAL, a character string variable is a packed array of characters, with integer subscripts starting at 1. The length of the string is established by the range of the array's subscripts, and is fixed. For example, if you define the variable NAME as a packed array of 30 characters, it can take on any 30-character string constant as its value.

2.4.2.2 **Records** - A record is a collection of related data items that may be of different types. Records are composed of fields, which are named by identifiers; each field contains a different data item. The data items can be of any type. To refer to a particular field of the record, you specify the record name and the field name, separated by a dot.

For example, you could define the record type Flight as follows:

Field		Contents
Flight = RECORD		
	Carrier : PACKED ARRAY [1..] OF CHAR;	Name of airline
	Flightno : PACKED ARRAY [1.] OF CHAR;	Integer flight number
	Depart,	Departure time
	Arrive : PACKED ARRAY [1..6] OF CHAR;	Arrival time
	Meals : CHAR;	Y or N indicating meal service
	END;	

To store all the information about a particular flight from Boston to Los Angeles, you might declare a variable named Boston_To_La of type Flight. To refer to the fields of this record, you would specify Boston_To_La.Carrier, Boston_To_La.Flightno, Boston_To_La.Depart, Boston_To_La.Arrive, and Boston_To_La.Meals.

2.4.2.3 **Files** - A file is a sequence of data items of the same type, called components of the file. The number of components in a file is not fixed; a file can be any length.

The components of a file can be any type except another file or a type containing a file. For example, you can define a file of arrays, a file of integers, a file of records, or a file of an enumerated type.

PASCAL defines the identifier TEXT to denote files with components of type CHAR. Text files are divided into lines so that you can read and write them line-by-line or character-by-character. The predefined files INPUT and OUTPUT are of type TEXT. These files refer to the standard input and output files, normally your terminal (in interactive mode) or the batch input and log file (in batch mode).

You can read or write a file through the file buffer variable, which is automatically created when you declare the file variable. The buffer variable is denoted by the file identifier followed by a circumflex.

For example, you could define the file variable Travel, with components of record type Flight, as described in Section 2.4.2.2 above. The buffer variable associated with this file would be denoted as Travel^, of type Flight.

The buffer variable takes on the value of one file component at a time. Predefined input procedures assign the value of a file component to the buffer variable; predefined output procedures assign the value of the buffer variable to a file component (see Chapter 7 for further information). Figure 2-1 illustrates the file buffer contents after a RESET procedure and after a read.

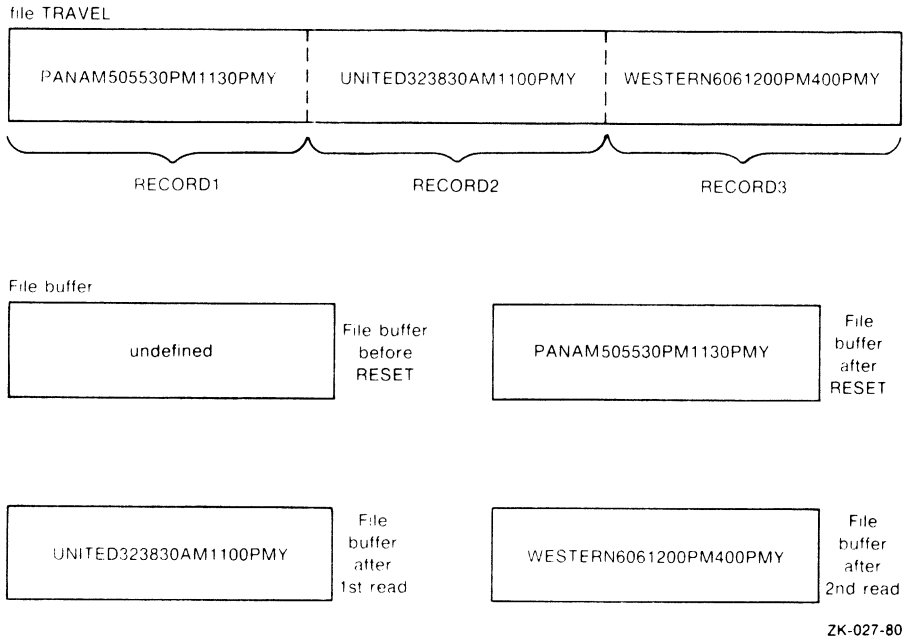


Figure 2-1 File Buffer Contents When Using READ and RESET

2.4.2.4 **Sets** - A set is a collection of scalar data items. Each set can have up to 256 elements with ordinal values from 0 to 255. The elements of the set must be specified within square brackets and separated by commas. For example, each of the following specifies a set with three elements:

```
[10, 20, 30]

[Motorcycle, Bicycle, Unicycle]

['a','b','c']
```

The empty set, which has no elements, is written as empty square brackets, [].

Because set elements must have ordinal values between 0 and 255, sets cannot contain integers outside this range. For example, the set [5..21, 27] is legal but [200..256] is not. Real numbers cannot be set elements.

2.4.3 Pointer Types

Normally, variables have the lifetime of the program or subprogram in which they are declared. That is, they exist only during execution of the declaring program or subprogram. Variables declared in the main program (program-level variables) are allocated in static storage and variables declared in a subprogram (subprogram-level variables) are allocated in stack storage. However, for some variables these lifetimes are inadequate. In addition, at times your program might require an unknown number of variables of a certain type. PASCAL allows you to use dynamic variables to meet these needs.

A dynamic variable is dynamically allocated when needed in the program. In contrast to other variables, dynamic variables are allocated in an area called heap storage.

Dynamic variables are not named by identifiers. Instead, you refer to them indirectly with pointers. Each pointer specifies the address of a dynamic variable of any type, called the base type.

To indicate a pointer type, you specify the name of a base type, preceded by a circumflex (^); for example ^person. To indicate the dynamic variable to which a pointer refers, you use the name of the pointer variable followed by a circumflex; for example pt^.

For example, dynamic variables of type Action can be referenced by pointers of type ^Action. If you declare the variables Ptr_Action1, Ptr_Action2, and Ptr_Action3 of type ^Action, you can use these pointers to refer to one or more dynamic variables of type Action. The value of each pointer variable is the address of a dynamic variable of type Action.

Using pointers to structured types, you can create linked lists. The NEW and DISPOSE procedures, described in Section 6.1.1.1, allocate and deallocate space for dynamic variables.

2.4.4 Type Compatibility

Type compatibility rules determine the operations and assignments that you can perform with data items of different types. This section provides a brief introduction to the concept of type compatibility and supplies some general rules. Specific rules are described with the operations and assignments to which they apply.

Two scalar types are compatible if their type identifiers are declared equivalent in the TYPE section (see Section 3.4). In addition, a subrange type is compatible with its base type, and two subranges of the same base type (or equivalent base types) are compatible.

For structured and pointer types, VAX-11 PASCAL enforces structural compatibility. Two structured or pointer types are compatible if their structures are identical. The way PASCAL determines structural compatibility depends on the types involved. For instance, the requirements for record compatibility differ from those for array compatibility. Chapter 4 covers specific compatibility requirements along with the description of each type.

PASCAL uses compatibility rules in the following three contexts:

- Expression compatibility
- Assignment compatibility
- Formal and actual parameter compatibility

Expression compatibility determines the types of operands you can use in an expression, as described in Section 2.5.

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

Formal and actual parameter compatibility determines the types of data you can pass in a parameter list. Value parameters follow the rules for assignment compatibility. VAR parameters follow somewhat different rules, as noted in Section 6.3.1.2.

2.5 EXPRESSIONS

An expression is a symbol or group of symbols that PASCAL can evaluate. These symbols can be constants, variables, or functions (see Chapter 6), or any combination of constants, variables, and functions, separated by operators. The simplest expression is a single variable or constant.

PASCAL provides the following types of operators:

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

2.5.1 Arithmetic Expressions

An arithmetic expression usually provides a formula for calculating a value. To construct an arithmetic expression, you combine numeric constants, variables, and function identifiers with one or more of the operators from Table 2-1.

Table 2-1
Arithmetic Operators

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

The addition, subtraction, multiplication, and exponentiation (+, -, *, and **) operators work on both integer and real values. They produce real results when applied to real values and integer results

when applied to integer values. If the expression contains values of both types, the result is a real number. The only exception to these rules concerns exponentiation. VAX-11 PASCAL defines the results of an integer raised to the power of a negative integer as follows:

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

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

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

The DIV and MOD operators apply to integer values only. DIV divides one integer by another, producing an integer result. DIV truncates the result, that is, it drops any fraction. It does not round the result. For example, the expression 23 DIV 12 equals 1 and -5 DIV 3 equals -1.

MOD is the modulus operator, which returns the remainder after dividing the first operand by the second. Thus, 5 MOD 3 evaluates to 2. Similarly, 3 MOD 3 evaluates to 0 and -4 MOD 3 evaluates to -1.

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

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

Table 2-2
Result Types for Arithmetic Expressions

First Operand	Second Operand	Operator			
		Multiply (*) Subtract (-) Add (+)	DIV MOD	Division (/)	Exponentiation (**)
Integer	Integer	Integer	Integer	Integer	Integer
	Real	Real		Real	Real
	Double	Double		Double	Double
Real	Integer	Real		Real	Real
	Real	Real		Real	Real
	Double	Double		Double	Double
Double	Integer	Double		Double	Double
	Real	Double		Double	Double
	Double	Double		Double	Double

2.5.2 **Relational Expressions**

A relational expression or condition tests the relationship between two arithmetic or logical expressions. A relational expression consists of two scalar or string variables or arithmetic expressions separated by one of the relational operators listed in Table 2-3.

Table 2-3
Relational Operators

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

Note that the two characters in the not equal (<>), greater than or equal (>=), and less than or equal (<=) operators must appear in the specified order and cannot be separated by a space.

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

2.5.3 Logical Expressions

Logical expressions test the truth value of combinations of conditions. A logical expression consists of two or more expressions that have Boolean results, separated by one of the logical operators in Table 2-4.

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 conditions to form a compound condition. The NOT operator reverses the truth value of a condition, so that if A is TRUE, NOT A is FALSE and vice versa.

As with relational expressions, the result of a logical expression is a Boolean value.

2.5.4 Set Expressions

You can use the operators in Table 2-5 with set variables, constants, and expressions.

Table 2-5
Set Operators

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

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

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

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

2.5.5 Precedence of Operators

The operators in an expression establish the order in which PASCAL evaluates the expression. Table 2-6 lists the order of precedence of the operators, from highest to lowest.

Table 2-6
Precedence of Operators

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

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

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

If you do not use parentheses, PASCAL will attempt to evaluate this expression as A <= (X AND B) <= Y and generates an error. The expression needs parentheses, as follows:

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

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

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

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

PASCAL evaluates the first expression according to the normal rules for precedence. First it multiplies 8 by 5 and divides the result (40) by 2. Then it subtracts 4 to get 16. The parentheses in the

second expression, however, force PASCAL to subtract before multiplying or dividing. Hence, it subtracts 4 from 2, getting -2. Then it divides -2 into 40, with -20 as the result.

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

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

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

2.6 SCOPE OF IDENTIFIERS

The scope of an identifier is the part of the program in which you have access to the identifier. In a PASCAL program, the scope of a constant, type, variable, or subprogram identifier is the block in which the identifier is declared. Figure 2-2 illustrates the scope of identifiers declared at various levels.

Declarations in the main program block specify global identifiers, which can be accessed in the main program and in all nested subprograms. For example, A and B in Figure 2-2 are global identifiers.

Declarations in subprogram blocks specify local identifiers. You can use a local identifier in the subprogram that contains its declaration and in all its nested subprograms. For example, the identifiers C and D are local to procedure Level1a and its nested subprograms Level2a and Level3a. You can use C and D in any of these subprograms, but not in the main program or in the subprograms Level1b, Level2b, and Level2c.

Similarly, local identifiers declared in Level1b are accessible to Level1b, Level2b, and Level2c, but not to Level1a, Level2a, Level3a, or the main program.

In general, once you define an identifier, it retains its meaning within the block containing its declaration. You can, however, redefine an identifier in a subprogram at a lower level. If you do so, the identifier assumes its new meaning only within the scope of the redefining block. Outside this block, the identifier keeps its original meaning. For example, B is declared at program level and redefined in Level2c. Within the scope of Level2c, B denotes a Boolean variable. Everywhere else in the program, however, B denotes an integer.

The identifiers accessible to each routine in Figure 2-2 are listed below.

Routine	Variables
Main program	A, B (integer)
Level1a	A, B (integer), C, D
Level2a	A, B (integer), C, D, E
Level3a	A, B (integer), C, D, E, F
Level1b	A, B (integer), G
Level2b	A, B (integer), G, H
Level2c	A, B (Boolean), J

```
PROGRAM Level0 (INPUT, OUTPUT);
        VAR A,B : INTEGER;
         .
         .
         .
        PROCEDURE Level1a (Z, Y);
            VAR C,D : INTEGER;
             .
             .
             .
            FUNCTION Level2a (X) : INTEGER;
                VAR E : REAL;
                     .
                     .
                     .
                   PROCEDURE Level3a (W);
                       VAR F : REAL;
                        .
                        .
                        .
                   END;  (*End procedure Level3a*)
             .
             .
             .
            END;  (*End function Level2a*)
         .
         .
         .
        END;  (*End procedure Level1a*)

        PROCEDURE Level1b (V, U, T);
            VAR G : INTEGER;
                 .
                 .
                 .
                PROCEDURE Level2b (S, R, Q);
                    VAR H : REAL;
                     .
                     .
                     .
                END;  (*End procedure Level2b*)

                PROCEDURE Level2c (P, O);
                    VAR B : BOOLEAN;
                        J : CHAR;
                     .
                     .
                     .
                END;  (*End procedure Level2c*)
         .
         .
         .
        END;  (* ..d procedure Level1b*)
 .
 .
 .
END.  (*End program Level0*)
```

ZK-072-80

Figure 2-2 Scope of Identifiers

CHAPTER 3

THE PROGRAM HEADING AND THE DECLARATION SECTION

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

The declaration section can contain the following sections:

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

Your program need not include all these sections, but the sections that are present must follow the order listed above. Although you can specify many labels, constants, types, variables, values, and subprograms, each section can appear only once per declaration section. Thus, you can use the reserved words LABEL, CONST, TYPE, VAR, and VALUE only once in each declaration section.

This chapter describes the program heading (Section 3.1), label declarations (Section 3.2), and constant definitions (Section 3.3). It also outlines type definitions (Section 3.4), variable declarations (Section 3.5), and value initializations (Section 3.6). Chapter 4 covers types, variables, and values in detail. Refer to Chapter 6 for information on procedures and functions.

PROGRAM

3.1 THE PROGRAM HEADING

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

Format

PROGRAM program-name [[(file-variable [[,file-variable ...]])]] ;

program-name

Specifies an identifier to be used as the name of the program.

e-variable

Specifies the identifier associated with an external file variable that the program uses.

The program name appears only in the heading and has no other purpose within the program. Because PASCAL treats the program name as a global identifier, it cannot be redefined at the program level.

The file variables listed in the program heading correspond to the external files that the program uses. The heading must include the names of all the external file variables. The predeclared text file variables INPUT and OUTPUT, by default, refer to your terminal (in interactive mode) or to the batch input and log files (in batch mode). You must declare file variables for all other external files in the main program declaration section, and specify those variables in the program heading. See Section 4.7 for more information on files.

Examples

1. PROGRAM Test1;

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

2. PROGRAM Squares (OUTPUT, INPUT);

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

3. PROGRAM Payroll (Employee, Salary, OUTPUT);

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

LABEL

3.2 LABEL DECLARATIONS

A label makes a statement accessible from a GOTO statement (see Section 5.6). The label section lists all the labels in the corresponding executable section.

Format

LABEL label[,label ...];

label

Specifies an unsigned integer. When you declare more than one label, you can specify them in any order.

A label can precede any statement in the program but can be accessed only by a GOTO statement. You must use a colon (:) to separate the label from the statement it precedes. Each label must precede exactly one statement within the scope of its declaration.

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

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

The GOTO statement in the procedure Max transfers control to the main program statement that has the label 75. However, you cannot use a GOTO statement in the main program to transfer control into the procedure at label 50. For further information, see Section 5.6, which describes the GOTO statement.

Example

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

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

CONST

3.3 CONSTANT DEFINITIONS

The constant section defines identifiers that represent constant values.

Format

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

constant-name

Specifies the identifier to be used as the name of the constant.

value

Specifies an integer, a real number, a string, a Boolean value, a enumerated type, or the name of another constant that is already defined.

Note that the value assigned to a constant identifier cannot be an expression. String values must be enclosed in apostrophes.

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

Examples

```
CONST Year = 1979;
      Month = 'January';
    Initial = 'p';
         Pi = 3.1415927;
      Tinyd = 1.7253D-10;
        Lie = FALSE;
    Untruth = Lie;
```

The CONST section specifies seven constant identifiers. Year, Pi, and Tinyd are integer, real, and double-precision numeric constants. Month represents a string value and Initial represents a character value. Both Lie and Untruth are equal to the Boolean value FALSE.

TYPE

3.4 TYPE DEFINITIONS

The type definition introduces the name and set of values for a type. Chapter 4 describes data types and includes examples of type definitions.

Format

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

type-identifier

Specifies the identifier to be used as the name of the type.

type-definition

Defines a type. The type can be:

- Predefined scalar (Section 4.1)
- Enumerated (Section 4.2)
- Subrange (Section 4.3)
- Array (Section 4.4)
- Record (Section 4.5)
- Set (Section 4.6)
- File (Section 4.7)
- Pointer (Section 4.8)

Note that you can use the identifier for a previously defined type in place of the type definition for a new type. In addition, you can define packed types for arrays, records, sets, and files, as described in Section 4.9.

VAR

3.5 VARIABLE DECLARATIONS

The variable declaration creates a variable and associates an identifier and a type with the variable. Chapter 4 describes data types and shows how to declare variables of each type.

Format

```
VAR variable-name  [[,variable-name...]]  :  type ;
  [[variable-name  [[,variable-name...]]  :  type ;...]]
```

variable-name

Specifies the identifier to be used as the name of the variable.

type

Names or defines a type. The type can be:

- Predefined scalar (Section 4.1)
- Enumerated (Section 4.2)
- Subrange (Section 4.3)
- Array (Section 4.4)
- Record (Section 4.5)
- Set (Section 4.6)
- File (Section 4.7)
- Pointer (Section 4.8)

You can also declare packed array, record, set, and file variables, as described in Section 4.9.

VALUE

3.6 VALUE INITIALIZATIONS

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

The description below presents general information on value initializations. The exact format of the value initialization depends on the type of variable being initialized. For detailed formats and examples, refer to the section in Chapter 4 that describes the type of the variable you need to initialize.

Format

```
VALUE  variable-name  := value ;
     [[variable-name  := value ;...]]
```

variable-name

Names the variable to be initialized. You cannot specify a list of variable names.

value

Specifies a constant of the same type as the variable, or specifies a constructor for an array or record variable.

You must specify a value of the correct type for each variable being initialized. You cannot specify an expression. Scalar variables require scalar constants and set variables require set constants. For arrays and record variables, you specify the value to be assigned to each element or field in a parenthesized list called a constructor (see Sections 4.4.3 and 4.5.2).

The VALUE initialization can appear only in the main program declaration section. You cannot initialize variables in procedures, functions, or modules.

CHAPTER 4

DATA TYPES

This chapter describes PASCAL data types, covers the use of the TYPE, VAR, and VALUE sections, and provides general information about how to use variables of each type. General formats and rules for the TYPE, VAR, and VALUE sections are provided in Chapter 3.

PASCAL provides two methods of declaring variables of a particular type. You can define the type in the TYPE section, and then use a declaration in the VAR section to declare one or more variables of the newly defined type. As described in Chapter 3, the general formats are as follows:

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

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

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

This chapter presents the format of the type definition for each PASCAL type. You can use the definition in either the TYPE or VAR section, as presented above. Examples of both methods appear throughout the chapter.

4.1 PREDEFINED SCALAR TYPES

PASCAL provides predefined types for integer, real, character, and Boolean data. Section 2.4.1.1 introduces these types. To declare a variable of a predefined scalar type, use a declaration in the VAR section.

Type Definition Format

```
INTEGER
REAL
SINGLE
DOUBLE
CHAR
BOOLEAN
```

Variables of type INTEGER can assume numeric values as described in Section 2.1.1. The values you assign to an integer variable must be of type INTEGER, of a type equivalent to INTEGER, or of an integer subrange type.

The identifiers REAL, SINGLE, and DOUBLE denote the real number types (see Section 2.1.2). REAL and SINGLE are synonymous; both denote single-precision real number values. The type DOUBLE allows you to declare double-precision real variables. You can assign real and integer values to a variable of type REAL, SINGLE, or DOUBLE. If you assign an integer value to a real variable, PASCAL converts the integer to a real value.

Variables of type CHAR are single characters from the ASCII character set. The values you assign to a CHAR variable must be of type CHAR or of a character subrange type.

Variables of type BOOLEAN can assume the values TRUE and FALSE. For assignment purposes, the type BOOLEAN is compatible with those variables and expressions that yield a Boolean result.

You can initialize a variable of a predefined scalar type with a constant of an assignment-compatible type, as shown in the example below. You cannot use an expression in a VALUE initialization.

Examples

```
VAR I, Temp : INTEGER;
    C : REAL;
    Cc : DOUBLE;
    Initial : CHAR;
    Answer : BOOLEAN;
    Grade : CHAR;

VALUE I := 0;
      Temp := 0;
      C := 5.713;
      Cc := 1.3D-5;
      Initial := 'P';
      Answer := TRUE;
```

The VAR section declares the variables I, Temp, C, Cc, Initial, Answer, and Grade. Note that you can group two or more variables of the same type in the VAR section, but you need not. For example, the integer variables I and Temp are declared together, but the character variables Initial and Grade are declared separately.

The VALUE section initializes six of the seven variables. You can specify the variables in any order; they need not follow their order of declaration. Unlike the VAR section, you cannot group variables in the VALUE section, even if the variables will be assigned the same value. Thus, the variables I and Temp must be initialized separately.

4.2 ENUMERATED TYPES

An enumerated type is an ordered set of values denoted by identifiers. To define an enumerated type, list in order all the identifiers denoting its values.

Type Definition Format

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

identifier

Specifies a constant value for the type.

The values of an enumerated type follow a left-to-right order, so that the last value in the list is greater than the first. For example:

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

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

The only restriction on the values of an enumerated type is that a value identifier cannot be defined for any other purpose. For example, the same TYPE section cannot also include the following:

```
Schoolyear = (Fall, Winter, Spring);
```

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

```
VALUE Quarter := Fall;
```

The variable Quarter takes on the initial value Fall.

Examples

```
TYPE Beverage = (Milk, Water, Cola, Beer);
     Sport = (Swim, Run, Ski);

VAR Cookie : (Oatmeal, Choc_Chip, Peanut_Butter, Sugar);
    Exercise, Fun : Sport;
    Drink : Beverage;

VALUE Cookie := Sugar;
      Fun := Ski;
```

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

The VAR section declares the variable Cookie, which can have the values Oatmeal, Choc_Chip, Peanut_Butter, and Sugar. The variables Exercise and Fun are declared of type Sport, and Drink is declared of type Beverage.

Initial values are established for the variables Cookie and Fun.

4.3 SUBRANGE TYPES

A subrange specifies a limited portion of another scalar type (called the base type) for use as a type.

Type Definition Format

```
lower-limit..upper-limit
```

lower-limit

Specifies the constant value at the lower limit of the subrange.

upper-limit

Specifies the constant value at the upper limit of the subrange.

The subrange type is defined only for the values between and including the lower and upper limits. The limits you specify must be constants; they cannot be expressions. Use the subrange symbol (..) to separate the limits of the subrange. The values in the subrange are in the same order as in the base type.

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

The use of subrange types can make a program clearer. For example, integer values for the days of the year range from 1 to 366. Any value outside this range is obviously incorrect. You could specify an integer subrange for the days of the year as follows:

```
VAR Day_Of_Year : 1..366;
```

By specifying a subrange, you indicate that the values of the variable Day_Of_Year are restricted to the integers from 1 to 366. If you use the Check option at compile time, the system generates a run-time error for an out-of-range assignment to a subrange variable. In this example, such an error occurs when an integer less than 1 or greater than 366 is assigned to Day_Of_Year.

Examples

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

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

VALUE Camp_Mos := Jul;
      First_Half := 'A';
```

This example defines the variables Camp_Mos and Leaf_Mos as subranges of the enumerated type Months. The variable First_Half is a subrange of the ASCII characters, with possible values uppercase A through uppercase M. The variable Word is a subrange of the integers from 0 to 65535. The VALUE section initializes Camp_Mos and First_Half. You initialize subrange variables in exactly the same way that you initialize variables of the base type.

4.4 ARRAY TYPES

An array is a group of elements of the same type that share a common name. You refer to each element of the array by the array name and a subscript (or index). An array type definition specifies the type of the subscripts and the type of the elements.

Type Definition Format

```
ARRAY [subscript-type[[,subscript-type...]] ] OF element-type
```

subscript-type

Specifies the type of the subscript. The subscript type can be an integer subrange, character, Boolean, or enumerated type, but not a real type,

element-type

Specifies the type of the elements of the array.

The elements of an array can be of any type. For example, you can define an array of integers, an array of records, or an array of real numbers. An array of arrays is a multidimensional array, as described in Section 4.4.1 below.

The subscripts of an array must be of a scalar type, but cannot be real numbers. Note that you cannot specify the type INTEGER as the subscript type. To use integer values as subscripts, you must specify an integer subrange. (An exception to this is dynamic array parameters; see Section 6.3.2.)

The range of the subscript type establishes the size of the array and how it is indexed. For example:

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

VAR Let1 :  Letters;
```

The array variable Let1 has 10 elements, referred to as Let1[1], Let1[2], Let1[3], and so on through Let1[10].

You can use array elements in expressions anywhere you can use variables of the element type. For the array as a whole, however, only the assignment (:=) operation is defined. An exception to this rule is character strings, as described in Section 4.4.2.

4.4.1 Multidimensional Arrays

An array with elements of an array type is a multidimensional array. An array can have any number of dimensions, and each dimension can have a different subscript type. For example, the following declares a 2-dimensional array variable:

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

PASCAL allows you to abbreviate the definition by specifying all the subscript types in one pair of brackets. For example:

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

To refer to an element of this array, you specify two subscripts, one integer and one character, in the order they were declared: Two_D[0,'A'], Two_D[0,'B'], and so on. You can also specify Two_D[0]['A']. The first subscript indicates the rows of the array and the second subscript indicates the columns. Figure 4-1 represents the array Two_D.

When you refer to the elements of Two_D, the first element in the first row is Two_D[0,'A']. The second element in this row is Two_D[0,'B']. The first element in the second row is Two_D[1,'A']. The last element in the last row is Two_D[4,'D']. In general, element j of row i is Two_D[I,J].

You can define arrays of three or more dimensions in a similar fashion. For example:

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

VAR Chess3d : ARRAY [1..3, 1..8, QR..KR] OF Chessmen;
```

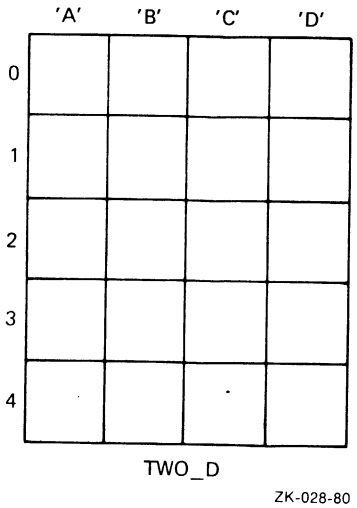


Figure 4-1 Two-Dimensional Array

This declaration specifies a 3-dimensional chess game. The indexes of the array are the levels, the ranks, and the files of the chessboard. For example, the reference Chess3d [1,1,qr] specifies the first level, first square in the upper left corner (bottom level, first rank, Queen's Rook file). Figure 4-2 illustrates the three levels of this array.

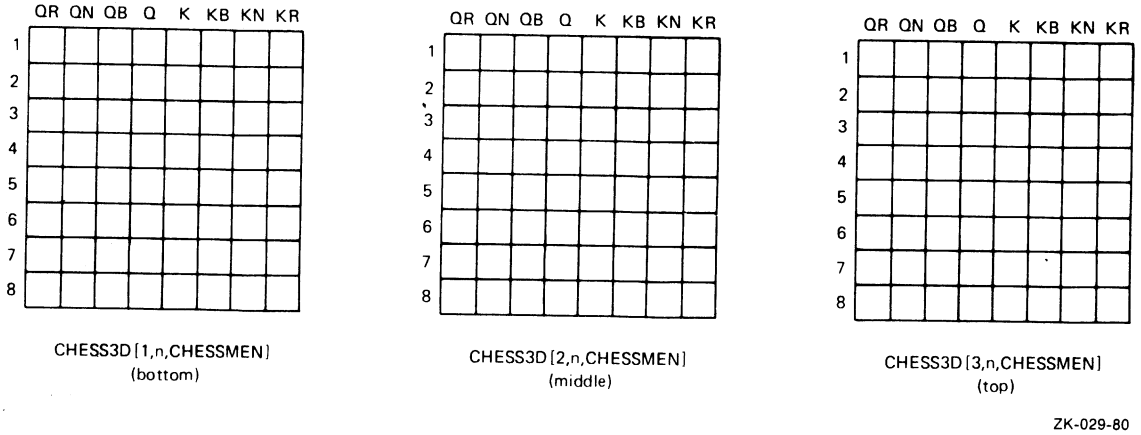


Figure 4-2 Three-Dimensional Array

When storing values in an array, PASCAL increments the first (leftmost) dimension slowest and the last (rightmost) dimension fastest. In the 3-dimensional array Chess3d, PASCAL starts by holding the first two subscripts constant while stepping through the values of Chessmen. Thus, the first values are assigned to elements Chess3d [1,1,QR] through Chess3d [1,1,KR].

Next, the second subscript is incremented and values are assigned to the elements Chess3d [1,2,QR] through Chess3d [1,2,KR]. After these eight elements are assigned, the second subscript is again incremented, and values are assigned to Chess3d [1,3,QR] through Chess3d [1,3,KR]. The assignment process continues with the first subscript held constant until the second subscript has been incremented from 1 to 8. Then the first subscript is incremented and the process is repeated. Hence, all values for the bottom level

(denoted by Chess3d [1,n,Chessmen]) are stored before any values for the middle level (denoted by Chess3d [2,n,Chessmen]). The top level (denoted by Chess3d [3,n,Chessmen]) receives its values last. Figure 4-3 illustrates this order.

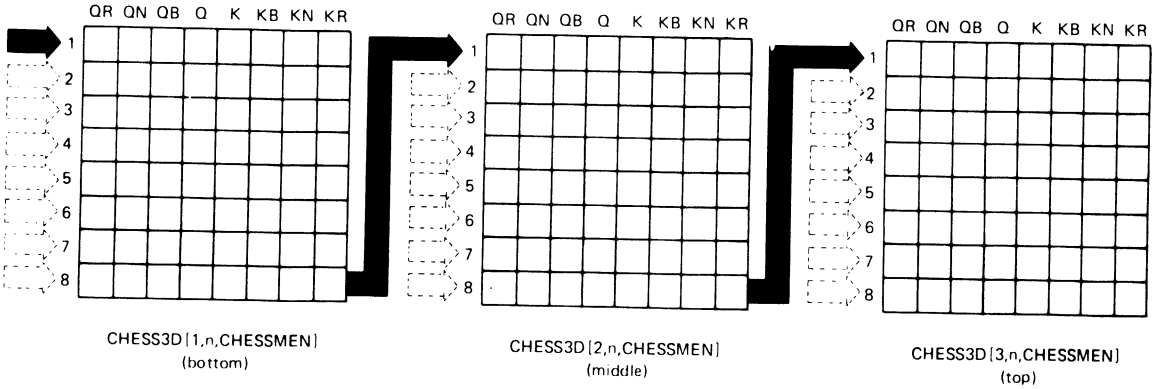


Figure 4-3 Storing Elements in an Array

4.4.2 String Variables

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

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

This declaration allows you to store a string of 20 characters in the array variable Name. The length of the string must be exactly 20 characters. PASCAL neither adds blanks to extend a shorter string nor truncates a longer string. If you specify a string of incorrect length, an error occurs.

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

```
'motherhood' > 'cherry pie'
```

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

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

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

4.4.3 Initializing Arrays

You can use a VALUE initialization in the following form to initialize an array with elements of any type except a file.

Format

```
VALUE variable-identifier := [[type-identifier]]([[n OF]] value, ... );
```

variable-identifier

Specifies the name of an array variable. You cannot initialize an array that has elements of a file type.

type-identifier

Specifies the type of the array being initialized. The type identifier is optional and is used only for documentation purposes.

n

Denotes an integer repetition factor that specifies the number of consecutive elements to be initialized with the next value.

value

Specifies a constant or constructor of the same type as the array elements.

A constructor is the actual values an array is to be initialized to. Each row of values is contained within parentheses. The values in the constructor must be constants; expressions are prohibited. You must specify a value for every element of the array. You cannot selectively initialize array elements.

To initialize a 1-dimensional array, specify the initial values in parentheses. For example, you can initialize a 5-element array with integers as follows:

```
VALUE My_Array := (1,2,3,4,5);
```

To initialize consecutive elements with the same value, use the repetition factor:

```
VALUE My_Array := (5 OF 1);
```

To initialize a 2-dimensional array, specify a constructor for each row, in parentheses. For example:

```
VAR Sun :  ARRAY [0..3, 1..5] OF REAL;

VALUE Sun := ((1.0, 1.1, 1.2, 1.3, 1.4), 2 OF (5 OF 0.0),
                        (10.1, 2 OF 11.0, 2 OF 11.1));
```

or

```
VALUE Sun := ((1.0, 1.1, 1.2, 1.3, 1.4), (0.0, 0.0, 0.0, 0.0, 0.0),
                        (0.0, 0.0, 0.0, 0.0, 0.0), (10.1, 11.0,
                        11.0, 11.1, 11.1));
```

PASCAL initializes array elements in the same order that it stores them. For example, Figure 4-4 shows the initial value assigned to each element of Sun.

	1	2	3	4	5
0	1.0	1.1	1.2	1.3	1.4
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0
3	10.1	11.0	11.0	11.1	11.1

ZK-031-80

Figure 4-4 Initial Values of 2-Dimensional Array

You can also initialize arrays of three or more dimensions. For example:

```
TYPE Cube = ARRAY[1..2, 1..3, 1..4] OF CHAR;

VAR Solid, Block :  Cube;

VALUE Solid := Cube (2 OF (3 OF (4 OF '*')));
```

This VALUE initialization assigns the asterisk character (*) to all elements of Solid. The type identifier Cube documents the type of this array for anyone who reads the declaration.

To initialize a string variable (a packed array of characters with a lower bound of 1), use the individual characters or a string constant. Suppose your program contains the following declaration:

```
VAR Pres :  PACKED ARRAY [1..10] OF CHAR;
```

You can initialize the array in either of the following ways:

```
VALUE Pres := ('J','E','F','F','E','R','S','O','N',' ');

VALUE Pres := 'Jefferson ';
```

4.4.4 Array Type Compatibility

You can assign one array to another only if the arrays are identical or compatible. Arrays of the same type or equivalent types are identical. For example:

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

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

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

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

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

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

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

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

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

4.4.5 **Array Examples**

1.
```
VAR Raceresults : ARRAY[1..50] OF Times;

VALUE Raceresults := (50 of 0);
```

This example declares the variable Raceresults as a 50-element array of Times. (Assume that Times is a numeric scalar type previously declared.) The VALUE declaration initializes Raceresults by assigning 0 to each element.

2.
```
TYPE Chessmen = (QR, QN, QB, Q, K, KB, KN, KR, P, E);

VAR Chess : ARRAY [1..8,QR..KR] OF Chessmen;

VALUE Chess := ((QR, QN, QB, Q, K, KB, KN, KR),
                (8 OF P), 4 OF (8 OF E),
                (8 OF P), (QR, QN, QB, Q, K, KB, KN, KR));
```

This example declares the type Chessmen and a 2-dimensional chessboard variable. The VALUE declaration initializes the board as it would be at the start of a game. The pieces from Queen's Rook (QR) to King's Rook (KR) are lined up along each end of the board, in the first and eighth rows of the array. The second and seventh rows of the array contain Pawns (P). The third through sixth rows are empty (E).

3.
```
TYPE String = PACKED ARRAY [1..10] OF CHAR;
VAR Composer, Word, Empty : String;
VALUE Word := 'engrossing';
      Composer := 'C.P.E.Bach';
      Empty := (10 OF ' ');
```

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

4.
```
CONST Days = 31;
TYPE  Weather = (Rain, Snow, Sunny, Cloudy, Foggy);
      Month = array [1..Days] of Weather;
```

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

4.5 RECORD TYPES

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

Type Definition Format

```
RECORD
[[ {field-identifiers : type ;[[field-identifiers : type...]];[[variant-clause ]]
   {variant-clause                                                              } ]]
END;
```

field-identifiers

Specifies the names of one or more fields. The names must be identifiers and separated by commas.

type

Specifies the type of the corresponding field(s). A field can be any type.

variant-clause

Specifies the variant part of the record. See Section 4.5.1.

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

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

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

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

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

```
Team_Rec.Wins := 9;

Team_Rec.Losses := 4;
```

Records can include fields that are themselves records. For example:

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

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

4.5.1 Records with Variants

A record variable can contain different kinds of information at different times if its type definition specifies one or more variants. A variant is a field or group of fields that can contain a different type or amount of data at different times during execution. Thus, two variables of the same record type can contain different types of data.

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

Format

```
CASE tag-field OF
    case-label-list : ([[field-identifiers : type ]][[ ;field-identifiers : type... ]])
      .
      .
      .
```

tag-field

Indicates the current variant of the record. You can specify the tag field in two ways:

1. tag-name : tag-type

   The tag field is a field in the record that is common to all variants. Tag-name and tag-type define the name and type of this field. The tag-type can be any scalar type

except a real type. You can use the tag field in the same way that you use any other field in the record, referring to using the record.fieldname format.

2. tag-type

   You must keep track of the currently valid variant. The tag type can be any scalar type except a real type.

case-label-list

Specifies one or more constants of the tag field type.

field-identifiers.

Specifies the names of one or more fields. The field names must be identifiers and must be separated by commas. Note that, instead of the field-identifiers, you can specify another variant clause, as in the last example in this section.

type

Specifies the type of the variant field.

When you specify the tag-field in the first form (tag-name : tag-type), you should reference only the fields in the currently valid variant. Thus, the value of the tag field must appear in the case label list that precedes the fields you are referencing. The following example shows the use of this form:

```
TYPE Stock = RECORD
             Part : 1..9999;
             Stock_Quantity : INTEGER;
             Supplier : Name;
             Case Onorder : BOOLEAN OF
                            TRUE :(Promised : Day;
                                   Order_Quantity : INTEGER;
                                   Price: real);
                            FALSE :(Last_Shipment : Day;
                                    Rec_Quantity : INTEGER;
                                    Cost : REAL)
             END;
```

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

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

```
TYPE Sex = (Female, Male);
     Hosp = RECORD
            Patient : Name;
            Birthdate : DATE;
            Age : INTEGER;
            CASE Sex OF
                 Female : (Births : 1..30);
                 Male : ()
            END;
```

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

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

```
TYPE Sex = (Female, Male);
     Hosp = RECORD
            Patient : Name;
            Birthdate : Date;
            Age : INTEGER;
            CASE Parsex : Sex of
                    Male : ();
                    Female : (CASE Births : BOOLEAN OF
                              FALSE : ();
                              TRUE : (Nofkids : INTEGER))
            END;
```

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

4.5.2 Initializing Records

To initialize a record variable, specify a value for each field of the record.

Format

```
VALUE variable-identifier :=[[type-identifier]](value [[, value...]] );
```

variable-identifier

Specifies the name of a record variable. The record cannot have a field of a file type.

type-identifier

Specifies the type of the record being initialized. The type identifier is optional and is used only for documentation purposes.

value(s)

Specifies a constant of the same type as the corresponding field.

The parenthesized list of values is called a constructor. In the constructor, the field values are specified in the order the fields are to appear in the record. If the record contains a variant, you must initialize the tag field and the corresponding variant fields. For example:

```
VAR Call : RECORD
           Caller : PACKED ARRAY [1..10] OF CHAR;
           Time : REAL;
           Subj : (Work, Play, Sales, Chat);
           CASE Return : BOOLEAN OF
                TRUE : (Hour : INTEGER);
                FALSE : ()
           END;

VALUE Call := ('Washington', 10.30, CHAT, TRUE, 12);
```

This initialization assigns a string constant to the Caller field, a real number to the Time field, and the identifier Chat to the Subj field. It supplies the Boolean value TRUE for the tag field Return, and initializes the variant field Hour with 12.

To initialize this record with a FALSE value for the tag field, you could specify the following:

```
VALUE Call := ('Washington', 10.30, Chat,FALSE);
```

This initialization assigns the same values as the previous VALUE initialization to all fields except the tag field. The tag field value is now last in the list because the FALSE case of Return specifies no additional fields.

You must always specify a value for the tag field, even if it has no tag name, to ensure that PASCAL initializes the correct variant.

4.5.3 Record Type Compatibility

Two records are compatible if their types are identical or equivalent. For example:

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

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

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

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

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

The following type is also compatible with Life and Plantlife:

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

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

If the records have variants, these criteria also apply:

- The records must have the same number of variants.
- Corresponding variants must have the same number of fields.
- Corresponding field types within corresponding variants must be compatible.
- The case labels associated with the variants must agree in number, but need not agree in value.

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

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

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

4.5.4 Record Examples

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

VAR   Fed, State, Local : Taxes;
VALUE Fed := (1979, 10000.0, 8000.0, 1500, FALSE, (5 OF 0.05));
```

This example declares and initializes the record Fed of type Taxes. The field Interest is initialized with a constructor within parentheses, because it is an array. Note that you can specify a repetition factor for the elements of an array, but not for the fields of a record.

2.
```
TYPE String = PACKED ARRAY [1..15] OF CHAR;
     Personal = RECORD
                  Name : String;
                  Address : RECORD
                            Number : INTEGER;
                            Street, Town : String;
                            Zip : 0..99999
                            END;

                  Age : 0..150
                  END;

VAR Faculty, Mascot, Student : Personal;
VALUE Faculty := ('Blaise Pascal  ',(1623, 'Pensees Street ',
                   'Clermont Alaska', 1662), 39);
```

The type Personal contains the field Address, which is of a record type. You must specify the values for this field in a constructor, nested within the constructor for the other fields. Without the parentheses enclosing the values 1623 to 1662, this declaration would cause an error at compile time.

4.6 SET TYPES

A set is a collection of data items of the same scalar type. The set type definition specifies the values that can be members of sets of that type.

Type Definition Format

SET OF base-type

base-type

Specifies the type from which the members of sets of this type are selected. You can use the identifier or definition of any scalar type, except a real type.

A set can have a maximum of 256 members, and the ordinal value of each member must be between 0 and 255. Therefore, real numbers cannot be set elements, nor can integers outside the range of 0 to 255.

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

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

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

To initialize a set in the VALUE section, specify a set constant in square brackets. For example:

```
VALUE Ski_Equip := [Boots..Goggles];
      Tennis_Equip := [Racquet, Shoes, Balls];
      Sleep_Equip := [] ;
```

Sets are compatible if their base types are identical or equivalent. For example:

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

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

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

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

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

Packing has no effect on set compatibility except when passing sets as VAR parameters (see Section 6.3.1.2). An unpacked set is compatible with a packed set if both sets meet the criteria above.

You can build set expressions by using the set operators described in Section 2.5.4. The set operators allow you to specify set intersection, difference, union, inclusion, and containment. In addition, you can assign a set expression to a set variable (see Section 5.2). The base type of the variable must include all members of the set to which the expression evaluates.

Examples

1.
```
TYPE Caps = SET OF CHAR;
VAR Vowel, Consonant : Caps;
VALUE Vowel := ['A', 'E', 'I', 'O', 'U'];
      Consonant := ['B'..'D', 'F'..'H', 'J'..'N', 'P'..'T', 'V'..'Z'];
```

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

2.
```
VAR PDP11S : SET OF 1..255;
VALUE PDP11S := [3, 4, 15, 20, 23, 34, 35, 40, 45, 55, 60, 70];
```

This example declares a set with an integer base type. The VALUE declaration specifies a set constant containing integer-valued members. Note that 780 cannot be a member of the set because its ordinal value is greater than 255.

4.7 FILE TYPES

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

Type Definition Format

```
FILE OF component-type
```

component-type

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

The arithmetic, relational, Boolean, and assignment operators cannot be used with file variables or structures containing file components. For example, you cannot assign one file variable to another, nor can you initialize a file variable. Likewise, you cannot compare two arrays that have file elements.

Type compatibility for files applies only to files that are passed to a subprogram. Two file parameters are compatible if their components are compatible and if both are packed or neither is packed. You can pass a file only as a VAR parameter. See Section 6.3.1.2 for more information on VAR parameters.

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

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

PASCAL creates Math_Scores^ as an integer buffer variable associated with the file Math_Scores. The buffer variable takes on the value of the file component at the current file position. The predeclared input and output procedures (see Chapter 7) move the file position, thus changing the value of the buffer variable. Figure 4-5 illustrates the contents of the file buffer during the use of the file Math_Scores.

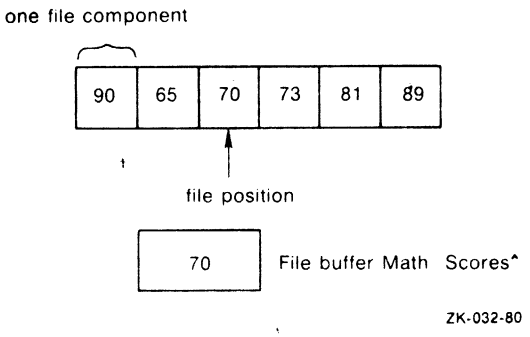


Figure 4-5 File Buffer Contents

Examples

1. VAR Truthvals : FILE OF BOOLEAN;

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

2. ```
   TYPE Names = PACKED ARRAY [1..20] OF CHAR;
        Data_File = FILE OF Names;
   VAR Accept_List, Reject_List, Wait_List : Data_File;
   ```

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

3.
```
VAR Results : FILE OF RECORD
                        Trial : INTEGER;
                        Date  : RECORD
                                Month : (Jan, Feb, Mar, Apr, May, Jun,
                                         Jul, Aug, Sep, Oct, Nov, Dec)
                                Day : 1..31;
                                Year : INTEGER
                                END;
                        Temp, Pressure : INTEGER;
                        Yield, Purity : REAL
                        END;
```

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

4.7.1 Internal and External Files

A file that is local to a program or subprogram is called an internal file. You can use an internal file only within the scope of the program or subprogram in which it is declared. (See Section 2.6 for rules of scope.) The system retains an internal file only during execution of the declaring program or subprogram. After execution the file is no longer accessible. The system creates a new file variable with the same name the next time it executes the declaring unit. The contents of the old file are not specified in the program heading. Only internal files can be components of structured types.

An external file exists outside the scope of the program in which it is declared. An external file can be created by the current PASCAL program, another PASCAL program, or a program written in another language. The system retains the contents of external file variables after the execution of the program. You must specify the names of external file variables in the program heading (Section 3.1). External files cannot be part of a structured type.

4.7.2 Text Files

A text file is a file with components of type CHAR. PASCAL defines a file type called TEXT as follows:

```
TYPE TEXT = FILE OF CHAR;
```

To declare a text file, specify a variable of type TEXT. For example:

```
VAR Poem : TEXT;
```

The text file variable Poem is a file of characters. For information on reading and writing text files, see Chapter 7.

Text files are divided into lines. Each line ends with a line marker separator. You can refer to the marker indirectly through the predeclared procedures READLN and WRITELN (Sections 7.2.3 and 7.3.6) and the predeclared function EOLN (Section 6.1.2).

The predeclared file variables INPUT and OUTPUT are files of type TEXT. These files are the defaults for all the predeclared text file procedures described in Chapter 7.

Examples

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

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

4.8 POINTER TYPES

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

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

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

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

Type Definition Format

```
^base-type-identifier
```

base-type-identifier

Indicates the type of the dynamic variable to which the pointer type refers. The base type can be any type.

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

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

M is a pointer variable bound to records of type Myrec. Specify M^ to denote the record variable to which M points.

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

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

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

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

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

VAX-11 PASCAL allows you to define pointers to types containing files. For example:

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

The pointer type X points to record type Y, which contains a file component in field Q. The files denoted by Q are not closed until execution of the program terminates. If you do not want the files to remain open throughout program execution, you must use the CLOSE procedure (Section 7.1.1) to close them. For example, to close the files defined in the TYPE section above, you must call CLOSE with the parameters X^.Q[1], X^.Q[2], and X^.Q[3].

You can initialize a pointer with the constant NIL as follows:

```
VALUE M := NIL;
```

As a result of the value initialization, the pointer variable M initially points to no variable. NIL is the only value you can specify to initialize a pointer.

Examples

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

     VAR Topten : ARRAY [1..10] OF Ptr_To_Hits;

     VALUE Topten := (10 OF NIL);
```

This example defines the record type Hits to which pointers of type Ptr_To_Hits refer. The array variable Topten has elements of the pointer type Ptr_To_Hits. Each element of the array is initialized with the constant Nil. The array Topten could be used in creating a linked list of 10 records of type HITS.

4.9 PACKED STRUCTURED TYPES

You can pack any of the structured types by specifying PACKED in the type or variable declaration. Packing means that the data items are stored as in as few bits as possible.

Type Definition Format

PACKED type-definition

type-definition

Defines an array, record, set, or file type. See Sections 4.4 through 4.7.

You can initialize all packed structures in the VALUE section in the same way that you initialize an unpacked structure of the same type. In general, packed data items require less storage space than unpacked data items of the same type. For specific information on the storage space allocated to packed and unpacked types, refer to the VAX-11 PASCAL User's Guide.

In PASCAL, a packed array of characters specifies a string variable. See Section 4.4.2.

Examples

1.
```
TYPE Ranges = PACKED RECORD
              Word : 0..65535;
              Byte : 0..32767;
              Bit : BOOLEAN
              END;
```

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

2.
```
VAR City_Census : PACKED ARRAY [1..25] OF 2500..50000;

VALUE City_Census := (25 OF 0);
```

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

CHAPTER 5

PASCAL STATEMENTS

VAX-11 PASCAL provides the following statements to perform actions within the program:

- Assignment statement
- CASE statement
- FOR statement
- GOTO statement
- IF-THEN statement
- IF-THEN-ELSE statement
- Procedure calls
- REPEAT statement
- WHILE statement
- WITH statement

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

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

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

  CASE

  IF-THEN

  IF-THEN-ELSE

- Repetitive statements:

  FOR

  REPEAT

  WHILE

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

PASCAL includes both simple and structured statements. The simple statements are the assignment and GOTO statements and the procedure call. The compound, conditional, repetitive, and WITH statements are

the structured statements. They enclose simple and structured statements that must be executed in order, repetitively, or when conditions are met. You can use a structured statement anywhere in the program that a simple statement is allowed. This manual uses the term statement to mean either a simple or a structured statement.

Compound Statement

5.1 THE COMPOUND STATEMENT

The compound statement allows for grouping PASCAL statements for sequential execution, as a single statement.

Format

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

statement

Denotes a simple or structured statement.

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

Assignment Statement

5.2 THE ASSIGNMENT STATEMENT

The assignment statement assigns a value to a variable.

Format

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

variable-name

Specifies the name of an array element, a file buffer variable, a function, a field of a record, a variable of any type except a file.

expression

Specifies a value, variable name, function reference, Boolean expression, set expression, or arithmetic expression.

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

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

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

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

For structured types, VAX-11 PASCAL enforces structural compatibility (see Section 2.4.4) in assignments. See also the description of the array, record, and set types in Chapter 4.

Examples

1. `X := 1;`

   The variable X is assigned the value 1.

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

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

3. `T := A<B;`

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

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

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

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

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

6. `My_Array := Your_Array;`

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

7. `Awardrec := New_Winner;`

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

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

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

Conditional Statements

5.3 CONDITIONAL STATEMENTS

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

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

CASE

5.3.1 The CASE Statement

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

Format

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

case-selector

Specifies an expression that evaluates to any scalar type except a real type.

case-label-list

Specifies one or more constants of the same type as the case selector, separated by commas.

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

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

If the value of the case selector expression does not match one of the case labels and you omit the OTHERWISE clause, the status of the CHECK run-time option determines the action that the system takes. If CHECK is enabled, the system prints an error message and terminates execution. If CHECK is not enabled, execution continues with the statement following the CASE statement. Refer to the VAX-11 PASCAL User's Guide for more information on the CHECK option.

Examples

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

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

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

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

3.
```
CASE  Alphabetic OF
      'A','E','I','O','U' : ALPHA_FLAG := Vowel;
                      'Y' : Alpha_Flag := Sometimes
      OTHERWISE Alpha_Flag := Consonant
END;
```

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

IF-THEN

5.3.2 The IF-THEN Statement

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

Format

```
IF expression THEN statement;
```

expression

Specifies a Boolean expression.

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

The THEN clause can specify a structured statement. Note, however, that if you use the compound statement, you must not place a semicolon between the words THEN and BEGIN. For example:

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

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

Examples

1.
```
IF ((X * 37/Constant) + Factor) > 1000.0 THEN
     ANSWER := ANSWER - FACTOR;
```

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

2.
```
IF (A>B) AND (B>C) THEN
  D := A-C;
```

If both relational expressions are true, D is assigned the value of A-C. Note that PASCAL does not always evaluate all the terms of a Boolean expression if it can evaluate the entire expression based on the value of one term. Thus, if A is less than or equal to B, the expression B>C may not be evaluated.

3.
```
IF  (Name = 'Smith') AND (Initial = 'J') THEN
     BEGIN
        Count := Count + 1;
        Smithadd[Count] := Address;
        WRITELN ('J Smith no.  ',Count, ' Lives At ', Address)
     END;
```

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

4.
```
IF Day = Thurs THEN
   FOR I := 1 TO Max_Emp DO
       Pay[I] := Salary[I] * (1-Tax_Rate-Fica);
```

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

IF-THEN-ELSE

5.3.3 The IF-THEN-ELSE Statement

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

Format

IF expression THEN statement1 ELSE statement2;

expression

Specifies a Boolean expression.

statement1

Denotes the statement to be executed if the expression is true.

statement2

Denotes the statement to be executed if the expression is false.

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

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

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

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

The variable D is assigned the value 1 if both A and B are equal to 1. An ELSE clause to be executed if A is not equal to 1 would be placed as follows:

```
IF A=1 THEN
   IF B<>1 THEN C:=1
   ELSE D:=1
ELSE C:=0;
```

Examples

1.
```
IF Disease
THEN
  WRITELN ('This person is sick.')
ELSE
  WRITELN ('This person is healthy.');
```

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

2.
```
IF Balance < 0.0 THEN
     BEGIN
       WRITELN ('Overdrawn by ',abs(BALANCE));
       WRITELN ('Loan of ',Loan,' at ',Rate,' % automatically deposited');
       Balance := Balance + Loan;
       Bill_Amt := Loan * (1+Rate)
     END
ELSE WRITELN ('No loan issued this month ');
WRITELN ('Balance is ',Balance);
```

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

Repetitive Statements

5.4 REPETITIVE STATEMENTS

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

- FOR statement
- REPEAT statement
- WHILE statement

FOR

5.4.1 The FOR Statement

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

Format

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

control-variable

Specifies the name of a variable of any scalar type except a real type.

initial-value

Specifies an expression of the same type as the control variable.

final-value

Specifies an expression of the same type as the control variable.

The control variable, the initial value, and the final value must all be of the same scalar type, but cannot belong to one of the real types. The repeated statements cannot change the value of the control variable.

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

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

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

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

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

When incrementing and decrementing the control variable, PASCAL uses units of the appropriate type. For numeric values, it adds or subtracts 1 upon each iteration. For values of other types, the control variable takes on each successive value of the type. For example, a control variable of type 'A'..'Z' is incremented (or decremented) by one character each time the loop is executed.

If the FOR loop terminates normally (that is, if the loop exits upon completion and not because of a GOTO statement, procedure call, or function call), the value of the control variable is left undefined. You cannot assume that it retains a value. Therefore, you must assign a new value to the control variable if you use it elsewhere in the program.

Examples

1.
```
FOR N := Lowbound TO Highbound DO
   Sum := Sum + Int_Array[N];
```

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

2.
```
FOR Year := 1899 DOWNTO 1801 DO
     IF (Year MOD 4) = 0 THEN
         WRITELN(Year:4,' IS A LEAP Year');
```

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

3.
```
FOR I := 1 TO 10 DO
    FOR J := 1 TO 10 DO
        A[I,J] := 0;
```

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

4.
```
FOR Employee := 1 TO N DO
     BEGIN
       Hrs := 40;
       FOR Day := Mon TO Fri DO
           IF Sick[Employee,Day] THEN
               Hrs := Hrs-8;
       Pay[Employee] := Wage[Employee] * Hrs
     END;
```

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

REPEAT

5.4.2 The REPEAT Statement

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

Format

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

expression

Specifies a Boolean expression.

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

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

Examples

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

Assume that the variable X is of type CHAR and the variables Digit_Count, Digit_Sum, and Char_Count are integers. The example reads a character (X) from the terminal. If X is a digit, the count of digits is incremented by 1 and the sum of digits is increased by the value of X. The ORD() function, described in Section 2.4.1.3, is used to compute the value of X. If X is not a digit, the variable Char_Count is incremented by one. The example continues processing characters from the terminal until it reaches an end-of-line condition.

5.4.3 The WHILE Statement

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

Format

```
WHILE expression DO statement;
```

expression

Specifies a Boolean expression.

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

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

Examples

1.
```
WHILE NOT EOF (File1) DO
  READLN (File1);
```

This statement skips to the end of the text file FILE1.

2.
```
WHILE NOT EOLN(INPUT) DO
  BEGIN
    READ(X);
    IF NOT (X IN ['A'..'Z','a'..'z','0'..'9']) THEN
       Err := Err + 1
  END;
```

This example reads an input character from the current line on the terminal. If the character is not a digit or letter, the error count (ERR) is incremented by 1.

3.
```
Sum := 0;
Ntests := 1;
Avg := 100;

WHILE  (Avg >= 90) AND (Ntests <= Maxtests) DO
    BEGIN

      Sum := Sum + Test [NTests];
      Avg := Sum Div Ntests;
      Ntests := Ntests +1
    END;
IF Avg < 90 THEN
WRITELN ('Your average dropped below 90 as of test ', Ntests:5);
```

After initializing Sum to zero, this program fragment repeatedly calculates a student's average test score. When the average score falls below 90, the calculations cease and the system prints an informational message.

WITH

5.5 THE WITH STATEMENT

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

Format

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

record-variable

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

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

Specifying more than one record variable has the same effect as nesting WITH statements. Thus, the following two statements are equivalent:

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

and

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

Note that if the record Dog is nested within the record Cat, you must specify Cat before Dog. The names must appear in the order of their declaration.

Examples

1.
```
VAR Taxes : RECORD
              Gross : REAL;
              Net : REAL;
              Bracket : REAL;
              Itemized : BOOLEAN;
              Paid : REAL
              END;
    .
    .
    .
WITH Taxes DO
  IF Net < 10000.0 THEN Itemized := TRUE;
```

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

2.
```
TYPE Name = PACKED ARRAY [1..20] OF CHAR;
     Date = RECORD
            Month : (Jan, Feb, Mar, Apr, May, Jun,
                     Jul, Aug, Sep, Oct, Nov, Dec);
            Day : 1..31;
            Year : INTEGER
            END;

VAR  Hosp : RECORD
            Patient : Name;
            Birthdate : Date;
            Age : INTEGER
            END;
   .
   .
   .
WITH Hosp, Birthdate DO
     BEGIN
       Patient := 'Thomas Jefferson    ';
       Month := Apr;
       Age := 236
     END;
```

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

GOTO

5.6 THE GOTO STATEMENT

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

Format

```
GOTO label;
```

label

Specifies a statement label.

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

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

Examples

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

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

This example shows how you can use the GOTO statement to exit from a loop. The loop computes the sum (Invertsum) of the inverses of the elements of Real_Array If one of the elements is 0, however, the sum is set to 0 and the GOTO statement forces an exit from the loop.

Procedure Call

5.7 THE PROCEDURE CALL

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

Format

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

procedure-name

Specifies the name of a procedure.

actual-parameter

Specifies a constant, an expression, the name of a procedure or function, or a variable of any type.

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

The formal parameter list in the procedure declaration determines the possible contents of the actual parameter list. The actual parameters must be compatible with the formal parameters. Depending on the types of the formal parameters, the actual parameters can be constants, variables, expressions, procedure names, or function names. An unsubscripted array name in a parameter list refers to the entire array. PASCAL passes actual parameters by the mechanism specified in the procedure declaration. See Section 6.3 and the VAX-11 PASCAL User's Guide for more information on parameters.

Examples

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

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

2. Taxes (Rate*Income, 'Pay');

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

3. HALT;

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

CHAPTER 6

PROCEDURES AND FUNCTIONS

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

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

VAX-11 PASCAL allows you to use several kinds of subprograms:

- Predeclared subprograms, described in Section 6.1.
- User-declared subprograms written in PASCAL. Sections 6.2 and 6.3 present the general format of subprograms and describe how parameters are passed to subprograms. Sections 6.4 through 6.6 describe how to declare PASCAL procedures and functions.
- External subprograms. This category includes subprograms written in other VAX-11 languages, VAX/VMS system service routines, and VAX-11 Run-Time Library procedures. Section 6.7 describes external subprograms; see the VAX-11 PASCAL User's Guide for additional information.

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

6.1 PREDECLARED SUBPROGRAMS

VAX-11 PASCAL provides predeclared procedures and functions that perform various commonly used tasks, such as input and output operations and mathematical functions. These predeclared subprograms are described in the following sections.

6.1.1 Predeclared Procedures

VAX-11 PASCAL declares procedures to perform input and output, allocate and destroy dynamic variables, supply the system date and time, pack and unpack array variables, and halt program execution. Table 6-1 summarizes these procedures.

Table 6-1
Predeclared Procedures

Procedure	Parameter Type	Action
CLOSE(f) [1]	f = file variable	Closes file f.
DATE(string)	string = variable of type PACKED ARRAY [1..11] OF CHAR	Assigns current date to string.
DISPOSE(p)	p = pointer variable	Deallocates storage for p^. The pointer variable p becomes undefined.
DISPOSE(p, t1,...,tn)	p = pointer variable t1,...,tn = tag field constants	Releases storage occupied by p^; used when p^ is a record with variants. Tag field values are optional; if specified they must be identical to those specified when storage was allocated by NEW.
FIND(f,n) [1]	f = file variable n = positive integer expression	Moves the current file position to component n of file f.
GET(f) [1]	f = file variable	Moves the current file position to the next component of f. Then GET(f) assigns the value of that component to f^, the file buffer variable.
HALT	None	Calls LIB$STOP, which signals SS$_ABORT. Without an appropriate condition handler, HALT terminates execution of the program.
LINELIMIT(f,n) [1]	f = text file variable n = integer expression	Terminates execution of the program when output to file f exceeds n lines. The value for n is reset to its default after each call to REWRITE for file f.
NEW(p)	p = pointer variable	Allocates storage for p^ and assigns its address to p.

1. Refer to Chapter 7 for descriptions of these input and output procedures.

(continued on next page)

Table 6-1 (Cont.)
Predeclared Procedures

Procedure	Parameter Type	Action
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,attributes) [1]	f = file variable attributes -- see Section 7.5	Opens the file f with the specified attributes.
PACK(a,i,z)	a = variable of type ARRAY [m..n] OF T i = starting subscript of array a z = variable of type PACKED ARRAY [u..v] OF T	Moves (v-u+1) elements from array a to array z by assigning elements a[i] through a[i+v-u] to z[u] through z[v]. The upper bound of a must be greater than or equal to (i+v-u).
PAGE(f) [1]	f = text file variable	Skips to the next page of file f. The next line written to f begins on the second line of a new page.
PUT(f) [1]	f = file variable	Writes the value of f^, the file buffer variable, into the file f and moves the current file position to the next component of f.
READ(f, v1,...,vn) [1]	f = file variable v1,...,vn = variables	For v1 through vn, READ assigns the next value in the input file f to the variable. You must specify at least one variable (v1). The default for f is INPUT.

1. Refer to Chapter 7 for descriptions of these input and output procedures.

(continued on next page)

Table 6-1 (Cont.)
Predeclared Procedures

Procedure	Parameter Type	Action
READLN(f, v1,...,vn) [1]	f = text file variable v1,...,vn = variables	Performs the READ procedure for v1 through vn, then sets the current file position to the beginning of the next line. The variable list is optional. The default for f is INPUT.
RESET(f) [1]	f = file variable	Enables reading from file f. RESET(f) moves the current file position to the beginning of the file f and assigns the first component of f to the file buffer variable, f^. EOF(f) is set to FALSE unless the file is empty.
REWRITE(f) [1]	f = file variable	Enables writing to file f. REWRITE(f) truncates the file f to zero length and sets EOF(f) to TRUE.
UNPACK(z,a,i)	z = variable of type PACKED ARRAY[u..v] OF T a = variable of type ARRAY [m..n] OF T i = starting subscript in array a	Moves (v-u+1) elements from array z to array a by assigning elements z[u] through z[v] to a[i] through a[i+v-u]. The upper bound of a must be greater than or equal to (i+v-u).
TIME(string)	string = variable of type PACKED ARRAY [1..11] OF CHAR	Assigns the current time to string.
WRITE(f,p1,...,pn) [1]	f = file variable p1,...,pn = write parameters	Writes the values of p1 through pn into the file f. At least one parameter (p1) must be specified. The default for f is OUTPUT.
WRITELN(f,p1,...,pn) [1]	f = text file variable p1,...,pn = write parameters	Performs the WRITE procedure, then skips to the beginning of the next line. The write parameters are optional. The default for f is OUTPUT.

1. Refer to Chapter 7 for descriptions of these input and output procedures.

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

NEW

The predeclared procedure NEW allocates memory for a dynamic variable. To refer to the dynamic variable, you must use a pointer variable (see Section 4.8 for a description of pointer types).

Format

```
NEW(p);
```

p

Specifies a pointer variable.

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

For example, you declare a pointer variable as follows:

```
VAR Ptr : ^INTEGER:
```

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

```
NEW(Ptr);
```

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

DISPOSE

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

Format

```
DISPOSE(p);
```

p

Specifies a pointer variable.

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

```
DISPOSE(Ptr);
```

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

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

Examples

```
PROGRAM LinkedList (INPUT, OUTPUT);

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

TYPE Student_Ptr = ^Student_Data;
     String = PACKED ARRAY[1..20] OF CHAR;
     Number = 1..99999;

     Student_Data = RECORD
                    Name : String;
                    Stud_Id : Number
                    Next : Student_Ptr
                    END;

VAR Start, Student : Student_Ptr;
    New_Id : Number;
    New_Name : String;
    Count : INTEGER;

PROCEDURE Write_Data(Student : Student_Ptr);

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

   VAR I,J : INTEGER;
       Next_Student : Student_Ptr;
   BEGIN
     WRITELN ('Name:', 'Student ID#:':29);
     REPEAT
       WRITELN(Student^.Name : 20, Student^.Stud_Id : 7);
       Next_Student := Student^.Next;
       DISPOSE (Student);
       Student := Next_Student
     UNTIL Student = NIL
   END;                                  (*End of Write_Data*)
```

```
(* Main Program *)

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

END.
```

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

```
NEW(Student);
```

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

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

```
DISPOSE(Student);
```

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

NEW and DISPOSE -- Record-With-Variants Form

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

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

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

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

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

For example, the following record represents a menu selection:

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

VAR Menu_Selection : Menu_Ptr;
```

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

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

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

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

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

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

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

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

6.1.1.2 **Miscellaneous Predeclared Procedures** - PASCAL provides the predeclared procedures PACK and UNPACK for packing and unpacking arrays. Packing means that the data items are stored as densely as possible.

PACK

You can declare arrays to be packed by specifying PACKED in the TYPE or VAR declaration (see Section 4.9). Sometimes, however, you might want to convert an array to a packed array within the executable section of the program. The predeclared procedure PACK copies elements of an unpacked array to a packed array.

Format

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

a,i,z

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

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

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

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

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

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

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

```
(*Declarations*)

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

(*Procedure call*)

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

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

UNPACK

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

Format

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

z,a,i

Same as for PACK.

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

For example, you cannot pass individual elements of a packed array to a subprogram as a VAR parameter (see Section 6.3.1.2). Therefore, you must unpack the array before you can pass its elements as VAR parameters:

```
(* Declarations *)

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

PROCEDURE Process_Elements (VAR Ch : CHAR);
   .
   .
   .

END;

(* Part of main program *)

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

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

DATE and TIME

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

Format

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

string

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

For example:

```
(* Declarations *)

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

(* Procedure calls *)

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

These two calls return results in the following format:

```
19-Jan-1957
14:20:25.98
```

The time is returned in 24-hour format. Thus, the time shown above is 14 hours, 20 minutes, 25 seconds, and 98 hundredths of a second. In

the DATE procedure, if the day of the month is a 1-digit number, the leading zero does not appear in the result, that is, a space appears before the date string.

6.1.2 Predeclared Functions

VAX-11 PASCAL provides predeclared functions to compute arithmetic values, test certain Boolean conditions, transfer data from one type to another, and perform other miscellaneous calculations. Predeclared functions return a value as a result. Table 6-2 summarizes the predeclared functions.

Arithmetic functions perform mathematical computations. Parameters to these functions can be integer, real, single-precision, or double-precision expressions. The ABS() and SQR() functions return a value of the same type as the parameter. All other arithmetic functions return a real value when the parameter is an integer, single-precision, or real value. When the parameter is a double-precision expression, the result is a double-precision value.

Boolean functions return Boolean values. The functions EOF() and EOLN() operate on files. EOF() tests for the end-of-file condition on a variable of any file type. EOLN() tests for the end-of-line condition on a text file variable. The ODD() function tests whether an integer parameter is odd or even. The UNDEFINED() function, for which you must supply a real value, returns the value TRUE if the parameter is "reserved." Variables containing a "reserved" value cause a VAX/VMS reserved operand fault when used in arithmetic computations. Refer to the VAX-11 Architecture Handbook for information about reserved values.

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

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

Table 6-2
Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
Arithmetic	ABS(x)	Integer, real, double	Same as x	Computes the absolute value of x.
	ARCTAN(x)	Integer, real	Real	Computes the arctangent of x.
		Double	Double	
	COS(x)	Integer, real	Real	Computes the cosine of x.
		Double	Double	

(continued on next page)

Table 6-2 (Cont.)
Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
Arithmetic (Cont.)	EXP(x)	Integer, real	Real	Computes e**x, the exponential function.
		Double	Double	
	LN(x)	Integer, real	Real	Computes the natural logarithm of x. The value of x must be greater than 0.
		Double	Double	
	SIN(x)	Integer, real	Real	Computes the sine of x.
		Double	Double	
	SQR(x)	Integer, real double	Same as x	Computes x**2, the square of x.
	SQRT(x)	Integer, real	Real	Computes the square root of x. If x is less than zero, PASCAL generates an error.
		Double	Double	
Boolean	..f)	File variable	Boolean	Indicates whether the file position is at the end of the file f. EOF(f) becomes TRUE only when the file position is after the last element in the file. The default for f is INPUT.
Boolean	EOLN(f)	Text file variable	Boolean	Indicates whether the position of file f is at the end of a line. EOLN(f) is TRUE only when the file position is after the last character in a line, in which case the value of f^ is a space. The default for f is INPUT.
	ODD(x)	Integer	Boolean	Returns TRUE if the nteger x is odd; eturns FALSE if x is even.
	UNDEFINED(r)	Real, double	Boolean	Returns TRUE if the value of is reserved operand; otherwise, returns FALSE.
Transfer	CARD(s)	Set	Integer	Returns the number of elements currently belonging to the set s.

(continued on next page)

Table 6-2 (Cont.)
Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
Transfer (Cont.)	CHR(x)	Integer	Char	Returns the character whose ordinal number is x (if it exists).
	ORD(x)	Any scalar type except a real type	Integer	Returns the ordinal (integer) value corresponding to the value of x.
	ROUND(x)	Real, double	Integer	Rounds the real or double-precision value x to the nearest integer.
	SNGL(d)	Double	Real	Rounds the double-precision real number d to a single-precision real number.
	TRUNC(x)	Real, double	Integer	Truncates the real or double-precision value x to an integer.
Miscellaneous	CLOCK	None	Integer	Returns an integer value equal to the central processor time used by the current image. The time is expressed in milliseconds.
	EXPO(r)	Real, double	Integer	Returns the integer-valued exponent of the floating-point representation of r.
	LOWER(a,n)	a = array variable n = integer constant	Subscript type of the nth dimension of a	Returns the lower bound of the nth dimension of array a. The parameter n is optional; if omitted, its default is 1. You usually use LOWER to determine the lower bound of a dynamic array parameter.
	PRED(x)	Any scalar type except a real type	Same type as parameter	Returns the predecessor value in the type of x (if a predecessor exists).
	SUCC(x)	Any scalar type except a real type	Same type as parameter	Returns the successor value in the type of x (if a successor exists).
	UPPER(a,n)	a = array variable n = integer constant	Subscript type of the nth dimension of a	Returns the upper bound of the nth dimension of array a. The parameter n is optional; if omitted, its default is 1. Like LOWER, UPPER is most commonly used with dynamic array parameters.

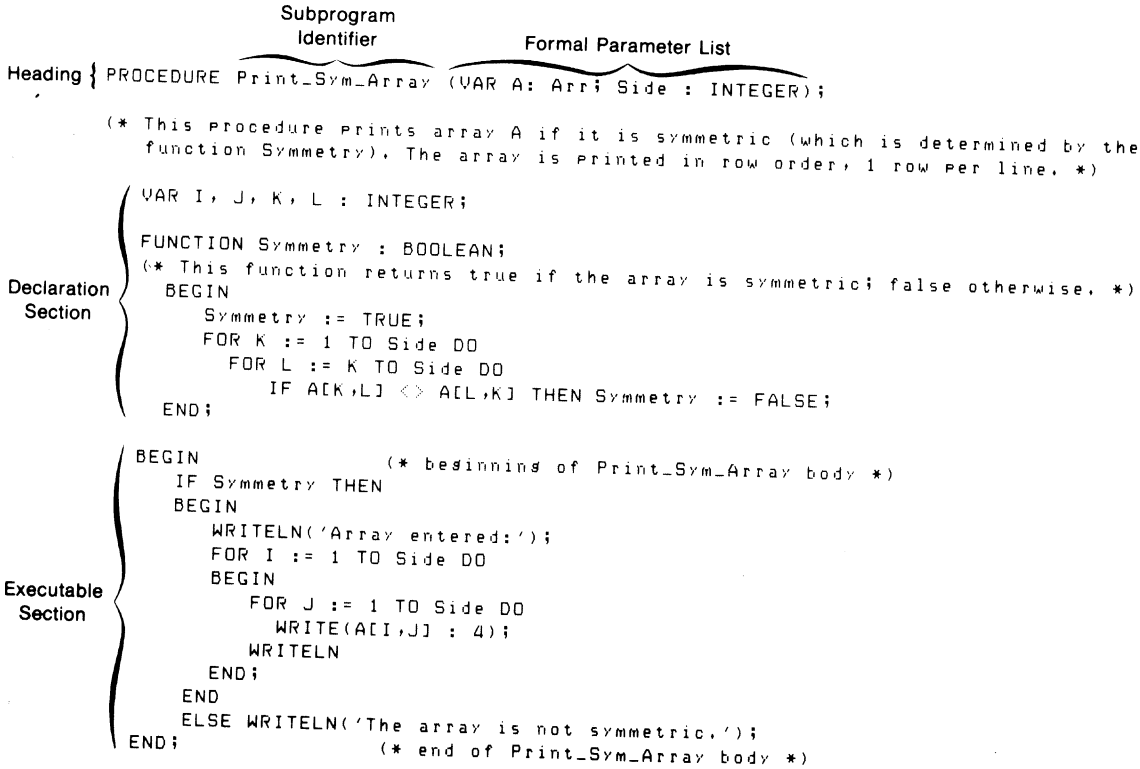
6.2 FORMAT OF A SUBPROGRAM

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

Figure 6-1 illustrates an example of a subprogram, the procedure Print_Sym_Array.

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

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



ZK-033-80

Figure 6-1 Sample Subprogram

Subprograms can be nested within other subprograms. For example, in Figure 6-1 the function SYMMETRY is nested in the procedure Print_Sym_Array. The order of nesting determines the scope of an identifier (see Section 2.6 for a discussion of scope).

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

6.3 PARAMETERS

Subprograms communicate data with the main program and with each other by means of parameters. A subprogram can have as many as 255 parameters, but need not have any at all.

The subprogram heading lists the formal parameters, which specify the type of data that will be passed to the subprogram. For example, in Figure 6-1, the formal parameter list for the procedure Print_Sym_Array is the following:

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

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

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

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

This procedure call passes the variable Current_Arr and the value of Current_Side to Print_Sym_Array. See Section 5.7 for information on procedure calls.

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

6.3.1 Format of the Formal Parameter List

The formal parameter list specifies the passing mechanism, name, and type of each formal parameter.

Format

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

mechanism-specifier

Indicates how PASCAL passes data to this parameter. The mechanism specifiers for PASCAL subprograms are VAR, PROCEDURE, and FUNCTION. The formal parameter list for external subprograms not written in PASCAL can also include the specifiers %IMMED, %DESCR, %STDESCR, %IMMED PROCEDURE, and %IMMED FUNCTION.

identifier-list

Specifies one or more identifiers, separated by commas.

type

Specifies the type of the parameters in the list. You can pass values, variables, procedures, and functions to a subprogram written in PASCAL, as described below. The type is a parameter list and is either a type identifier or a dynamic array type.

PASCAL semantics provide two methods for passing parameters to PASCAL subprograms.

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

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

For external subprograms not written in PASCAL, VAX-11 PASCAL provides the mechanism specifiers %IMMED, %DESCR, %STLDESCR, %IMMED PROCEDURE, and %IMMED FUNCTION. These specifiers force the use of mechanisms defined in the VAX-11 procedure calling standard. They allow you to pass immediate values, scalar and array descriptors, string descriptors, procedure identifiers, and function identifiers to external subprograms. See the VAX-11 PASCAL User's Guide for information on their use.

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

The actual parameter corresponding to a formal value parameter must be a compatible expression. Value parameters follow the rules for assignment compatibility as described in Sections 2.4.4 and 5.2. For example, the following list passes all parameters by value:

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

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

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

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

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

In Figure 6-1, the formal parameter list includes an example of a VAR parameter:

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

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

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

```
(VAR Sea, Breeze : REAL;  Wind : INTEGER;  VAR Sick : Med_File)
```

As a result of this formal parameter list, the actual parameters corresponding to Sea, Breeze, and Sick are passed as VAR parameters, and the actual parameter corresponding to Wind is passed as a value parameter (the default).

Compatibility

Variables passed to a subprogram as actual VAR parameters must be compatible with the corresponding formal parameters. In VAX-11 PASCAL, formal and actual parameter compatibility is the same as assignment compatibility (described in Section 2.4.4) with the following exceptions:

- You cannot pass an integer actual parameter to a real formal parameter.
- You cannot pass an element of a packed structure with the VAR specifier, although you can pass the entire structure. You must unpack the structure or assign its elements to scalar variables before you can pass individual elements.
- You cannot pass a variable of a packed set type to a formal parameter that is an unpacked set type, and vice versa.

6.3.1.3 **Formal Procedure and Function Parameters** - To pass a PASCAL procedure or function as a parameter, use the PROCEDURE or FUNCTION mechanism specifier. A subprogram that is passed as a parameter must have only value parameters. It must not require VAR parameters.

For example, the following formal parameter list specifies a formal function parameter:

```
(FUNCTION Operation :  REAL;  Addends :  REAL)
```

This list specifies a function parameter and a value parameter. The corresponding actual parameter list must contain both the name of a real-valued function and a real expression. The actual function parameter corresponding to Operation can have only value parameters in its formal parameter list.

For example, suppose you had declared the following functions:

```
FUNCTION Sum (A:REAL): REAL;
    .
    .
    .

FUNCTION Quotient (A, B: REAL; VAR R : REAL): REAL;
    .
    .
    .
```

Sum can be legally passed to the formal parameter Operation declared above. However, because Quotient contains the VAR parameter R, it cannot be passed to Operation.

You can also specify formal procedure parameters. For example:

```
(Err : INTEGER ; PROCEDURE Printer)
```

This specification allows you to pass a procedure to the subprogram that contains this formal parameter list. The actual procedure parameter corresponding to Printer can have only value parameters.

6.3.2 Dynamic Array Parameters

Some programming applications require general subprograms that can process arrays with different bounds. VAX-11 PASCAL allows you to declare subprograms with dynamic array parameters. You can call the subprogram with arrays of different sizes, as long as their bounds are within those specified by the formal parameter.

For example, you could write a procedure that sums the elements of a 1-dimensional array. Each time you use the procedure, however, you might want to pass arrays with different bounds. Instead of declaring multiple procedures using arrays of each possible size, you can use a dynamic array parameter. The procedure will treat the formal parameter as if its bounds were those of the actual parameter.

In subprograms that contain dynamic array parameters, you use the predeclared functions LOWER() and UPPER() to return the lower and upper bounds of the actual array parameter.

The format of a dynamic array parameter is:

```
array-identifier-list : [PACKED] ARRAY [subscript-type-identifier
                        [,subscript-type-identifier...]] OF type-identifier;
```

Note that you must use a type identifier to specify the range of subscripts. You cannot use a subrange. The type identifier can be any of the predefined scalar types (for example, INTEGER) except one of the real types.

The elements and subscripts of the actual and formal dynamic array parameters must be of compatible types. The rules for dynamic array compatibility are the same as those for other arrays (Section 4.4.4) with one exception: the range of the subscript types of the actual array parameter must be within the range specified for the formal parameter.

Examples

1.
```
PROGRAM Dynarr(INPUT, OUTPUT);
(* This program illustrates the use of dynamic array parameters.
The procedure Sum is called from the main program with two
different actual parameters -- Arr1 and Arr2. *)

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

```
PROCEDURE Sum (VAR Inarr : ARRAY [RNG] OF INTEGER);
(* This procedure accepts actual array parameters with integer
elements whose subscripts are within the range specified by
RNG. *)

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

(* Main Program *)
BEGIN
  WRITELN ('Type 5 Integers');
  FOR K:= 1 TO 5 DO
    READ (Arr1[K]);
  SUM (Arr1);
  WRITELN('Type 14 Integers');
  FOR J:= 7 TO 20 DO
    READ(Arr2[j]);
  SUM (Arr2)
END.
```

This program sums the elements of the arrays Arr1 and Arr2. The procedure Sum includes a 1-dimensional dynamic array parameter, Inarr, whose subscripts are of type Rng. Within the main program, Sum is called with two different arrays: Arr1 with subscript type [1..5] and Arr2 with subscript type [7..20].

The first procedure call, Sum(Arr1), passes Arr1 to Sum. The predeclared functions LOWER() and UPPER() are used here to specify the lower and upper bounds of the actual parameter corresponding to Inarr. Thus, LOWER(Inarr) evaluates to 1 and UPPER(Inarr) evaluates to 5. The FOR loop processes array elements Inarr[1] to Inarr[5]. When Sum is called with Arr2, the For loop processes the elements Inarr[7] to Inarr[20].

2.
```
TYPE Level_Range =  1..6;
     Nclasses =  1..8;
     Nstudents =  1..40;
     Names =  PACKED ARRAY [1..35] OF CHAR;
  .
  .
  .
PROCEDURE Kid_Count (School :  ARRAY [Level_Range,
                    Nclasses, Nstudents] OF Names);
  .
  .
  .
```

This example defines School as a 3-dimensional dynamic array parameter. Each array passed to School might contain the names of all the students in a particular elementary school. The subscripts of the array denote the number of grades in the school, the number of classes at each grade level, and the number of students in each class.

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

6.4 DECLARING A PROCEDURE

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

To declare a procedure, specify its header and block in the procedure and function section. You can declare a procedure in the main program, in a module (see Section 6.8), or in another subprogram.

Format

```
PROCEDURE procedure-identifier [[(formal-parameter-list)]];

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

BEGIN
 [[statement[[;statement...]] ]]
END;
```

or

```
PROCEDURE procedure-identifier (formal-parameter-list); directive;
```

procedure-identifier

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

formal-parameter-list

Contains the mechanism specifiers, names, and types of the formal parameters (see Section 6.3.1).

label-section

Declares local labels.

constant-section

Defines local constant identifiers.

type-section

Defines local types.

variable-section

Declares local variables.

procedure-and-function-section

Declares local procedures and functions.

statement

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

directive

Specifies a FORWARD, FORTRAN, or EXTERN procedure (see Sections 6.6 and 6.7). If you use a directive, the procedure declaration must not include a block.

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

- The declaration section cannot contain VALUE initializations.
- The procedure block ends with END followed by a semicolon (;), rather than a period (.) as in the main program.

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

Examples

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

```
CONST Number = 20;
TYPE Range =  0..100;
     List = ARRAY[1..Number] OF Range;

VAR Arr : List;
    Minimum, Maximum : Range;
    Average : REAL;
```

1.
```
PROCEDURE Read_Write (VAR A : List);
   VAR I : INTEGER;
   BEGIN
     WRITELN ('Type a list of 20 integers, in the range of 0 to 100.');
     FOR I := 1 TO Number DO
     BEGIN
       READ(A[I]);
       WRITE(A[I]:7);
       WRITELN
     END
   END;
```

The procedure Read_Write reads a list of 20 integers, inserts them into the array A, and writes the array. Read_Write uses one Var parameter -- the array A.

Given the declaration of Arr, the following is a valid procedure call:

```
Read_Write(Arr);
```

As a result of this call, the list of integers is written in the array Arr. The value of this array is then returned to the program unit that called the procedure Read_Write.

2.
```
PROCEDURE Min_Max_Avg (VAR Min, Max : Range; VAR Avg : REAL    : List);
   VAR Sum, NMax, NMin, J : INTEGER;

   BEGIN
     Max := A[1];  Min := Max;  Sum := Max;
     NMax := 1;  NMin := 1;
     FOR J := 2 TO Number DO
     BEGIN
       Sum := Sum + A[J];
       IF A[J] > Max THEN
       BEGIN Max := A[J];
             NMax := 1
       END
       ELSE IF A[J] = Max THEN
             NMax := NMax + 1;
       IF A[J] < Min THEN
       BEGIN Min := A[J];
             NMin := 1
       END
       ELSE  IF A[J] = Min THEN
             NMin := NMin + 1
     END;
     Avg := Sum/Number;
     WRITELN;
     WRITELN('Maximum =',Max:4,', occurring',NMax:4, ' times');
     WRITELN;
     WRITELN('Minimum =', Min:4,', occurring', NMin:4,' times');
     WRITELN;
     WRITELN ('Average value (truncated) =', TRUNC(Avg):10);
     WRITELN ('Average value =', Avg : 20)
END;
```

This procedure computes the minimum, maximum, and average values in array A. It also counts the number of times the minimum and maximum values occurred and stores those numbers in NMin and NMax. The WRITELN statements print the results of each of these computations.

Min, Max, and Avg are formal VAR parameters. Their values are returned to the calling program unit and can be used for further computations in the program. A is specified as a value parameter because its value does not change in the procedure.

The following is a valid procedure call to the procedure:

```
Min_Max_Avg(Minimum, Maximum, Average, ARR);
```

The values of the formal parameters Min, Max, and Avg are returned to the actual parameters Minimum, Maximum, and Average.

6.5 DECLARING A FUNCTION

A function is a group of statements that compute a scalar or pointer value. To declare a function, specify its heading and block in the procedure and function section.

Format

```
FUNCTION function-identifier [[(formal-parameter-list)]] :result-type;

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

    BEGIN
     [[statement[[;statement...]] ]]
    END;
```

or

```
FUNCTION function-identifier  [[(formal-parameter-list)]]: result-type;  directive;
```

function-identifier

Specifies the identifier to be used as the name of the function.

formal-parameter-list

Contains the mechanism specifiers, names, and types of the formal parameters (see Section 3.1).

result-type

Specifies the type of the function's result. The result must be a scalar or pointer value.

label-section

Declares local labels.

constant-section

Defines local constant indentifiers.

type-section

Defines local types.

variable-section

Declares local variables.

procedure-and-function-section

Declares local procedures and functions.

statement

Specifies an action to be performed by the function. A function can contain any of the statements described in Chapter 5. It must contain a statement that assigns a value to the function identifier (for every potential path through the code). If it does not, the value of the function could be undefined.

directive

Specifies a FORWARD, FORTRAN, or EXTERN function (see Sections 6.6 and 6.7). If you use a directive, the function declaration must not include a block.

A function consists of a heading and a block. The formal parameter list in the function heading is identical in format to the one in the procedure heading (Refer to Section 6.3). The function block is similar in structure and contents to the main program, with the following exceptions:

- The function cannot contain a value initialization section.
- The function block ends with END followed by a semicolon (;), rather than a period (.) as in the main program.

You must declare all variables that are local to the function, but you should not redeclare a variable with the same name as a formal parameter or the function identifier.

Each function must include a statement or statements that assign a value of the result type to the function name. The last value that is assigned to the function name is returned to the calling program unit. To use the value, include a function call in the calling unit. Unlike a procedure call, a function call is not a statement. It simply represents a value of the function's result type.

Side Effects

A side effect is an assignment to a nonlocal variable, or to a VAR parameter, within a function block. Side effects can change the intended action of a program and therefore should be avoided. The following program illustrates an example of a side effect.

```
PROGRAM Example (OUTPUT);
VAR X,Y : INTEGER;
    Ans1, Ans2 : BOOLEAN;

   FUNCTION Positive (ThisVar : INTEGER) : BOOLEAN;
   BEGIN
     Positive := FALSE;
     IF ThisVar > 0 THEN
     BEGIN
        X := ThisVar - 10;                    (* Side effect on X *)
        Positive := TRUE
     END
   END;  (* end Positive*)

BEGIN                                         (* MAIN PROGRAM *)
   Y := 7;  X :=15;
   ANS1 := Positive(X) AND Positive(Y);
   WRITELN ('Ans1 equals',Ans1);
   Y := 7; X := 15;
   Ans2 := Positive(Y) AND Positive(X);
   WRITELN ('Ans2 equals',Ans2)
END.
```

This example may generate the following output:

```
Ans1 equals          TRUE
Ans2 equals          FALSE
```

Thus, the output depends on which function call is evaluated first: Positive(Y) or Positive(X). PASCAL does not guarantee which part of an expression is evaluated first. The resulting value of a function should not depend on when the function is called, as it does in the example above. Therefore, you should avoid side effects on global variables.

Examples

1.
```
FUNCTION Coupons :  REAL;
    VAR Ans : (Yes, No);
        Amount, Subt : REAL;
    BEGIN
      Subt := 0;
      WRITELN ('Any coupons? Type yes or no.');
      READLN (Ans);
      IF Ans=Yes THEN
          BEGIN
            WRITELN ('Type value of each coupon, one per line.');
            WRITELN ('Press CTRL/Z when finished.');
            REPEAT
              READLN (Amount);
              Subt := Subt + Amount
            UNTIL EOF
          END;
      Coupons := Subt
    END;                (* End Coupons *)
```

The function Coupons computes the total value of a group of coupons. It uses only the three local variables, Ans, Amount, and Subt, and requires no parameters. The result of this function is the real total of the coupon values. The assignment statement, Coupons := Subt, assigns the result to the function identifier.

To use the function Coupons, specify its name, as follows:

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

The function call is treated as a real-valued expression in this statement. Note that you can use the function call in the same way that you can use a value of its result type.

2.
```
FUNCTION Symmetry (VAR A : Arr) : BOOLEAN;
(*This function returns TRUE if the array A is symmetric, else
FALSE is returned *).

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

The function Symmetry uses one VAR parameter, the array A. Symmetry returns a Boolean value -- the result is TRUE if A is symmetric and FALSE if A is not symmetric.

6.6 FORWARD DECLARATIONS

Normally, you must declare subprograms before you refer to them. However, a subprogram can reference another subprogram that has not yet been declared if you use a forward declaration. The forward declaration provides the compiler with information about the forward-declared subprogram's formal parameters and indicates that the block of the subprogram follows later in the source file.

For example, a complete declaration is impossible if two subprograms call each other recursively. Omitting the declaration is also impossible because PASCAL needs information about a subprogram's formal parameters before it can compile a reference to the subprogram. Therefore, you must forward-declare one of the recursive subprograms.

A forward declaration consists of the subprogram heading (including the formal parameter list, if any, and the result type if it is a function) and the FORWARD directive, without a subprogram block. For example:

```
PROCEDURE Chestnut (Bld :REAL; Doc : CHAR; VAR Arc : Rec); FORWARD;
```

This example forward-declares the procedure Chestnut. The forward declaration includes only the information shown in the example.

When you specify the block of a forward-declared subprogam, you must not repeat the formal parameter list or the result type of a function. Except for these omissions, declare the heading and block in the normal way.

Example

```
FUNCTION Adder (Op1, Op2, Op3 : REAL) : REAL; FORWARD;

PROCEDURE Printer (Student : Name_Array) ;
  .
  .
  .

  BEGIN
  .
  .
  .

        Z := Adder (A,B,C)
  END;

FUNCTION Adder (* Op1, Op2, Op3 : REAL : REAL*);
  .
  .
  .

  BEGIN
  .
  .
  .

     Printer ('Leonardo da Vinci');
  .
  .
  .

  END;
```

This example forward-declares the function Adder. The block of the function appears after the declaration of the procedure Printer. Note that the heading of the Adder block specifies its formal parameters and result type within comment delimiters. Although you must omit the parameter list and result type when you define the function block, inserting them as a comment is a good documentation practice.

6.7 EXTERNAL SUBPROGRAMS

The FORTRAN and EXTERN directives indicate procedures and functions external to a PASCAL program. With either of these directives, you can declare subprograms written in another language (such as FORTRAN or MACRO), PASCAL subprograms that are compiled separately, VAX/VMS system service routines, and VAX-11 Run-Time Library routines. In VAX-11 PASCAL, the FORTRAN and EXTERN directives are equivalent. However, to ensure portability of your program, you should use FORTRAN only for external routines written in FORTRAN.

If you declare separately compiled PASCAL subprograms as EXTERN, their names must be unique. That is, no two outermost level PASCAL subprograms can have the same name. In addition, an external subprogram cannot have the same name as the main program.

The %IMMED, %DESCR, %STDESCR, %IMMED PROCEDURE, and %IMMED FUNCTION mechanism specifiers can be used only with external subprograms that are not written in PASCAL. See the VAX-11 PASCAL User's Guide for details.

Examples

1. FUNCTION MTH$TANH (Angle : REAL) : REAL; EXTERN;

   The function MTH$TANH is a VAX-11 Run-Time Library routine. This declaration declares MTH$TANH as an external subprogram.

2. PROCEDURE FORSTRING(%STDESCR S : PACKED ARRAY[INTEGER] OF CHAR); FORTRAN;

   This declares the FORTRAN procedure FORSTRING. The formal parameter list specifies S as a dynamic array parameter that is passed by string descriptor.

[illegible] SEPARATE COMPILATION

By placing PASCAL procedures and functions in a module, you can compile them separately from the main program. At link time, you specify the compiled files containing the main program and the modules to be included in the executable image. The executable image can include any number of modules, and each module can contain any number of subprograms.

Format

```
MODULE module-name [(program-parameters)];
[constant-section;]
[type-section;]
[variable-section;]
procedure-and-function-section;
END.
```

module-name

Specifies the identifier to be used as the name of the module.

program-parameters

Lists the external files. This list must be identical in order and in content to the list in the main program heading.

constant-section

Declares global constant identifiers exactly as in the main program.

type-section

Defines global types exactly as in the main program.

variable-section

Declares global variables exactly as in the main program.

procedure-and-function-section

Declares the procedures and functions contained in the module.

A module is similar to a main program, except that it has no value initialization section and no executable section. Modules can contain constant, type, variable, and procedure and function sections. (PASCAL issues a warning-level message if a module contains label declarations, but ignores the labels.) The constant, type, and variable sections and the program parameters must be identical to those in the main program. The procedure and function section defines the subprograms contained in the module.

To ensure that the program parameters and the constant, type and variable sections are identical in the main program and all modules, you can place them in a separate VAX/VMS file. Then you can use the %INCLUDE directive to insert the contents of the file into the main program and all modules, instead of repeating all the declarations and definitions. The %INCLUDE directive is explained in Section 1.6.

Subprograms declared at the outermost level of a module can be declared and called from the main program (or another module). You must declare the subprogram with the EXTERN modifier in the calling program unit. Similarly, subprograms declared at the outermost level of the main program can be declared as EXTERN in a module.

Each subprogram in the module can access data declared either locally or by the main program.

Examples

```
MODULE SEP (INPUT, OUTPUT);
VAR I :  INTEGER;
PROCEDURE Reader (N : INTEGER);
   VAR K,P : INTEGER;
   BEGIN
     I := 0;
     FOR K := 1 TO N DO
         BEGIN
            READ (P);
            IF P=0 THEN I := I + 1;
         END
   END;  (* Reader *)
   END.  (*MODULE Sep *)
```

The Module SEP contains one procedure, Reader. You can declare Reader as an external subprogram in another module or in the main program. Note that the external files and the VAR sections of the module and main program must be identical.

CHAPTER 7

INPUT AND OUTPUT

PASCAL provides predefined procedures and functions to perform input and output to file variables. These procedures and functions are divided into the following categories:

General Procedures and Functions

- CLOSE -- closes an internal or external file
- EOF() -- indicates the end of an input file
- EOLN() -- indicates the end of an input line
- FIND -- performs direct access to file components
- OPEN -- opens a VAX/VMS file with specified characteristics

Input Procedures

- GET -- reads a file component into the file buffer variable
- READ -- reads a file component into a specified variable
- READLN -- reads a line from a text file
- RESET -- opens a file and prepares it for input

Output Procedures

- LINELIMIT -- terminates program execution after a specified number of lines have been written into a text file
- PAGE -- advances output to the next page of a text file
- PUT -- writes the file buffer variable into the specified file
- REWRITE -- truncates a file to length zero and prepares it for output
- WRITE -- writes specified values into a file
- WRITELN -- writes a line into a text file

Chapter 6 introduced predefined functions and procedures. This chapter describes input and output routines, general procedures and functions, and terminal I/O.

General Procedures and Functions

7.1 GENERAL PROCEDURES AND FUNCTIONS

This section describes the following general procedures:

- CLOSE
- EOF()
- EOLN()
- FIND
- OPEN

CLOSE

7.1.1 The CLOSE Procedure

The CLOSE procedure closes an open file.

Format

```
CLOSE (file-variable);
```

file-variable

Specifies the file to be closed.

Execution of this procedure causes the system to close the file and, if the file is internal, delete it. Each file is automatically closed upon exit from the block in which it is declared.

You cannot close a file that has not been opened either explicitly by the OPEN procedure or implicitly by the RESET or REWRITE procedure. The predeclared file variables INPUT and OUTPUT cannot be closed. If you attempt to close a file that was never opened an error occurs.

Examples

```
CLOSE (Albums);
```

This procedure closes the file Albums to further access, and deletes the file if it is an internal file.

EOF( )

7.1.2 The EOF() Function

The EOF() (end-of-file) function indicates whether the file pointer is positioned after the last element in the file.

Format

EOF[(file-variable)]

file-variable

Specifies the name of the input file variable. The default value is INPUT.

The Boolean function EOF() returns a TRUE result when the file pointer is positioned after the last element in the file.

The EOF() function returns a FALSE result even when the last component of the input file is read into the file buffer. You must attempt to read another file component to determine whether the file pointer is positioned at end-of-file.

If you attempt to read a file after EOF becomes true an error results.

Examples

1.
```
Coupons := 0;
WHILE NOT EOF DO
   BEGIN
     READLN(Coupon_Amount);
     Coupons := Coupons + Coupon_Amount
   END;
```

This example calculates the total value of the coupons contained in the file INPUT. The loop is performed until the EOLN function returns the value TRUE.

2.
```
IF NOT EOF DO
BEGIN
  WHILE NOT EOLN DO
    BEGIN
      READLN (Master_File);
      If New_Customer <> YES
      THEN
        OLD=OLD+1
      ELSE
        NEW=NEW+1
     END;
```

EOLN( )

7.1.3 The EOLN() Function

The EOLN() (end-of-line) function tests for the end of a line marker within a text file.

Format

EOLN[[(file-variable)]]

file-variable

Specifies the text file variable name. The default value is INPUT.

The Boolean function EOLN() returns a TRUE result when the file pointer is positioned after the last character in a line.

The EOLN() function returns a FALSE result when the last component on the line is read into the file buffer. You must attempt to read another character to determine whether the file pointer is positioned at end-of-line. If you use the EOLN() function on a non-text file an error occurs.

Examples

1.
```
Num_Chars := 0;
WHILE NOT EOLN DO
  BEGIN
    READ (CH);
    Num_Chars := Num_Chars + 1;
  END;
READLN;
```

This example assumes that a new line of input is being scanned, and calculates the number of characters in that line.

The WHILE statement continues to execute until after the end-of-line marker is read.

2.
```
WHILE NOT EOF(Master_File) DO
BEGIN
  WHILE NOT EOLN(Master_File) DO
    BEGIN
      READ (Master_File, X);
      IF NOT (X IN ['A'..'Z','a'..'z','0'..'9']) THEN
        Err := Err + 1
    END;
  READLN(Master_File)
END;
```

This example scans the characters on each line of a text file called Master_File and checks for characters that are not digits or letters. If a nondigit or nonletter character is encountered in the file, the counter Err is increased. The loop is executed until the last component in the file is read.

FIND

7.1.4 The FIND Procedure

The FIND procedure positions a file at a specified component.

Format

```
FIND (file-variable, integer-expression);
```

file-variable

Specifies a file that is open for direct access. The file must have fixed-length records.

integer-expression

Specifies a positive integer expression indicating the component at which to position the file. If the component number is a zero or a negative an error occurs.

The FIND procedure allows you to directly access the components of a file. The file must be open for direct access. That is, you must have specified DIRECT in the OPEN statement for that file. In addition, the file must have fixed-length records.

You can use the FIND procedure to move forward or backward in a file.

After execution of the FIND procedure, the file is positioned at the specified component. The file buffer variable assumes the value of the component. For example:

```
FIND (Albums, 4);
```

As the result of this statement, the file position moves to the fourth component in the file Albums. The file buffer variable Albums^ assumes the value of the fourth component.

If you specify a component beyond the end of the file, an error occurs.

You can use the FIND procedure only when reading a file that was opened by the OPEN procedure. If the file is open for output (that is, with REWRITE), a call to FIND results in a run-time error.

Examples

1.
```
FIND (Albums, Current + 2);
```

   If the value of Current is 6, this statement causes the file position to move to the eighth component. The file buffer variable Albums^ assumes the value of the component.

2.
```
FIND (Albums, Current - 1);
```

   If the value of Current is 6, this statement causes the file position to move backward one component to the fifth component. The file buffer variable Albums^ assumes the value of the fifth component.

OPEN

7.1.5 The OPEN Procedure

The OPEN procedure opens a file, defines file access mode and allows you to specify file attributes. You can list the parameters in a positional default order, or by specifying the parameter name and parameter value.

Formats

```
1.   OPEN( file-variable  [[, file-name]] [[, history]]
     [[,record-length]] [[, record-access-method]] [[, record-type]]
     [[,carriage-control]] );

       or

2.   OPEN(FILE_VARIABLE:=file-variable
          [[,FILE_NAME:=file-name]]
          [[,HISTORY:=file-status]]
          [[,RECORD_LENGTH:=positive integer]]
          [[,RECORD_ACCESS_METHOD:=record-access-mode]]
          [[,RECORD_TYPE:=record-type]]
          [[,CARRIAGE_CONTROL:=carriage-control]] );
```

file-variable

Specifies the PASCAL file variable associated with the file being opened. You cannot open the predeclared file variable INPUT.

file-name

Provides information about the file for the operating system. You can use a character string literal, a string constant name, a string variable containing a VAX/VMS file specification or a logical name (see the VAX-11 PASCAL User's Guide for more information). You can use a constant or a variable which has been defined as a logical name or a file specification. Apostrophes are required to delimit a character string literal used as the file name or a logical name. If you omit the file specification, PASCAL uses the default values shown in Table 7-1.

Table 7-1
Default Values for VAX/VMS File Specifications

Element	Default
Node	Local computer
Device	Current user device
Directory	Current user directory
File name	PASCAL file variable name or logical name
File type	DAT
Version number (history)	OLD: highest current number NEW: highest current number + 1

The file variable and file-name parameters designate the file to be opened. The remaining parameters are summarized in Table 7-2. Except for the file variable, all parameters are optional. If the parameter names (such as RECORD_TYPE) are not used, as in format 1, the parameters must be in the specified order. If parameter names are used, as in format 2, the parameters can be specified in any order.

You can mix the use of positional default and parameter name. But once a parameter name is used, all the following parameters values must have parameter names in that OPEN statement.

Table 7-2
Summary of File Attributes

Parameter Name	Parameter Values	Default
History	OLD or NEW	NEW (OLD, if the file is opened using RESET)
Record_length	Any unsigned integer	133 bytes
Record_access_method	DIRECT or SEQUENTIAL	SEQUENTIAL
Record_type	FIXED or VARIABLE	VARIABLE for new file; for old file, record type established at file creation
Carriage_control	LIST, CARRIAGE, FORTRAN NOCARRIAGE, NONE	LIST for all text files; NOCARRIAGE for all other files. Old files use their existing carriage control attributes

The OPEN procedure opens a file for access by the program. If the file does not exist and has a HISTORY of NEW, a file is created with the specified name and attributes. You cannot use OPEN on a file variable that is already open, or on the predeclared file variable INPUT.

Because the RESET and REWRITE procedures implicitly open files, you need not always use the OPEN procedure. RESET and REWRITE impose the defaults for VAX/VMS file specification, record length, record access method, record type, and carriage control shown in Tables 7-1 and 7-2. For the file status attribute, RESET uses a default of OLD, and REWRITE uses a default of NEW. You must use the OPEN procedure for the following:

- To create a file with fixed-length records
- To open a file for DIRECT access
- To specify for a text file a maximum record length other than 133

7.1.5.1 **History -- NEW or OLD** - The file history indicates whether the specified file exists or must be created. A file history of NEW indicates that a new file must be created with the specified attributes. NEW is the default value.

A file history of OLD indicates that an existing file is to be opened. An error occurs if the file cannot be found. OLD is invalid for internal files, which are newly created each time the declaring program or subprogram is executed.

7.1.5.2 **Record Length** - This unsigned integer specifies for a text file the maximum record size in bytes. The default value is 133 bytes. For a text file with variable-length records, the record length value denotes the maximum number of characters on a line. The record length parameter applies only to text files. For a nontext file, this parameter has no meaning and is ignored.

By default, text files have variable-length records. If you create a text file with fixed-length records, the record length specifies the exact length of the records. A file is read with the record length specified at its creation.

7.1.5.3 **Record Access Method -- SEQUENTIAL or DIRECT** - The record access method specifies sequential or direct access to the components of the file. In SEQUENTIAL mode, you can access files with fixed- or variable-length records. The default access mode is SEQUENTIAL.

The DIRECT mode allows you to use the FIND procedure to gain random access to RMS sequential files with fixed-length records. You cannot access a file with variable-length records in DIRECT mode. Direct access files can be accessed for random reading only.

7.1.5.4 **Record Type -- FIXED or VARIABLE** - The record type specifies the structure of the records in a file. A value of FIXED indicates that all records in the file have the same length. A value of VARIABLE indicates that the records in the file can vary in length. VARIABLE is the default for a new file. For an existing file, the default is the record type associated with the file at its creation.

7.1.5.5 **Carriage Control -- LIST, CARRIAGE, FORTRAN, or NOCARRIAGE NONE** - The carriage control option specifies the carriage control format for a file. A value of LIST indicates single spacing between components. LIST is the default for all text files, including the predefined file OUTPUT.

The CARRIAGE or FORTRAN options indicate that the first character of every output line is a carriage control character.

NOCARRIAGE or NONE options specifies a file with no carriage control.

You cannot create a file with one carriage control format and later read it with another.

7.1.5.6 **Examples**

1.
```
VAR Userguide : TEXT;
  .
  .
  .
OPEN (Userguide);
```

When the OPEN procedure is executed, the system first attempts to use Userguide as a logical name. If no such logical name is assigned, the system creates the file Userguide.Dat in your default device and directory on the local computer. By default, the file is created with a record length of 133 bytes and records of variable length. The system then opens the file for sequential access.

2.
```
OPEN (Albums, 'DB1:[EASTWEST]INVENT', ACCESS_METHOD := DIRECT,
HISTORY := OLD);
```

This example opens the existing VAX/VMS file DB1:[EASTWEST]INVENT.DAT for direct access. The VAX/VMS file is known to the PASCAL program as the file variable Albums. The order of the OPEN statement for this file has been changed by using parameter names.

3.
```
OPEN (Solar, 'Energy', HISTORY := NEW, RECORD_TYPE := FIXED);
```

Assuming that Energy is defined as a logical name, this statement creates a file with the VAX/VMS specification designated by the logical name Energy. The file is created with fixed-length records.

Input Procedures

7.2 INPUT PROCEDURES

This section describes the following input procedures:

- GET
- READ
- READLN
- RESET

GET

7.2.1 The GET Procedure

The GET procedure reads the next component of a file into the file buffer variable.

Format

```
GET (file-variable);
```

file-variable

Specifies the file to be read.

Before the GET procedure is used to read one or more file components, you must first call the RESET procedure to prepare the file for reading (input). RESET moves the file position to the first component and assigns its value to the file buffer variable.

As a result of the GET procedure, the file position moves to the next component of the file. Unless the EOF marker is encountered, the file buffer variable takes on the value of that component. For example:

```
RESET (BOOKS);
NEWREC := BOOKS^;
GET (BOOKS);
```

After execution of the RESET procedure, the file buffer variable Books^ is equal to the first component of the file. The assignment statement assigns this value to the variable Newrec. The GET procedure then assigns the value of the second component to Books^, moving the file position to the second component. The next GET procedure moves the file position to the third component, as shown in Figure 7-1.

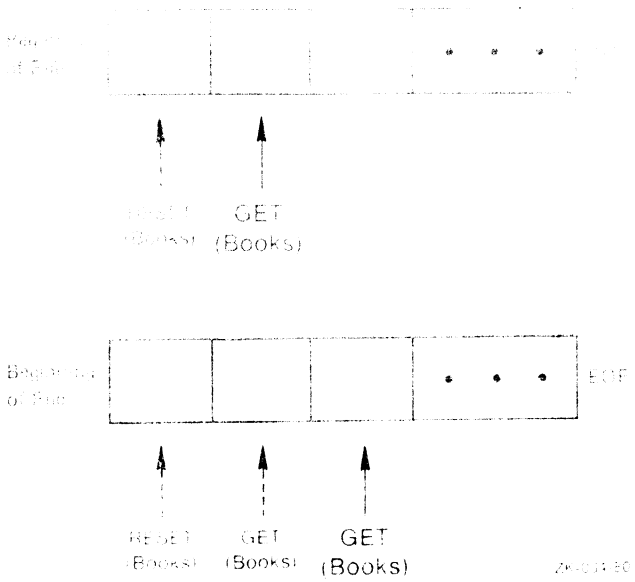


Figure 7-1 File Position after GET

By repeatedly using GET statements, you can read sequentially through the file.

When you reach the end of the file, EOF() automatically becomes TRUE and the file buffer variable becomes undefined. If GET() is used when EOF() is true PASCAL signals a run-time error and program execution is aborted.

Examples

```
GET (Phones);
```

This example reads the next component of the file PHONES into the file buffer variable Phones^. Prior to executing GET, EOF(Phones) must be FALSE; if it is TRUE, an error occurs.

READ

7.2.2 The READ Procedure

The READ procedure reads one or more file components into a variable of the component type.

Format

```
READ ([[file-variable,]] variable-name [[,variable-name...]] );
```

file-variable

Specifies the input file. If you omit the file variable, PASCAL uses INPUT by default.

variable-name

Specifies the variable into which the file component(s) will be read. For a text file, many file components can be read into one variable.

By definition, the READ procedure for a nontext file performs an assignment statement and a GET procedure for each variable name. Thus, the procedure call

```
READ (file-variable, variable-name);
```

is equivalent to

```
variable-name := file-variable^;
GET (file-variable);
```

The READ procedure reads from the file until it has found a value for each variable in the list. The first value read is assigned to the first variable in the list, the second value is assigned to the second variable, and so on. The values and the variables must be of compatible types.

For a text file, more than one file component (that is, more than one character) can be read into a single variable. For example, many text file components can be read into a string or numeric variable. The READ procedure repeats the assignment and GET process until it has read a sequence of characters that represent a value for the next variable in the parameter list. It continues to read components from the file until it has assigned a value to each variable in the list.

After the last character has been read from a line of a text file, EOLN() is TRUE and the file buffer variable contains a space. A call to READ at this point moves the file position to the beginning of the next line, and characters are read starting at that position.

Values from a text file can be read into variables of integer, real, character, string, and enumerated types. In the file, values to be read into integer and real variables must be separated by spaces, or tabs, or end-of-line markers. Values to be read into character variables, however, must not be separated because they are read literally, character-by-character. Constants of enumerated types must be separated by at least one space, tab, or end-of-line marker. If an invalid character is encountered in an identifier, the constant is terminated.

When Boolean input is read, any of the following is allowed: TRUE, FALSE, T, or F.

When identifiers of enumerated type are read from a text file, only the first 31 characters are used. However, you need only input sufficient characters to make the identifier unique within the identifier table. All lowercase letters are converted to uppercase letters.

You can use READ to read a sequence of characters from a text file into a variable of type PACKED ARRAY[1..n] OF CHAR. PASCAL assigns successive characters from the file to elements of the array, in order, until each element has been assigned a value. If any characters remain on the line after the array is full, the next READ begins with the next character on that line. If the end of the line is encountered before the array is full, the remaining elements are assigned spaces.

When a file is positioned at the end of a line, and a READ with string variable arguments is called, the file position moves to the beginning of the next line. Characters are then read into the specified variable. If this line is empty, the string is filled with spaces.

Every non-empty text file ends with an end-of-line marker and an end-of-file marker. Therefore, the function EOF() never becomes TRUE when you are reading strings with the READ procedure. To test EOF() when reading strings, use the READLN procedure to advance the file beyond the end-of-line marker.

Examples

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

   Assume that Temp, Age, and Weight are real variables, with the following values entered:

   ```
   98.6 11 75
   ```

   The variable Temp takes on the value 98.6, Age takes on the value 11.0, and Weight takes on the value 75.0. Note that you need not type all three values on the same line.

2. ```
   TYPE String = PACKED ARRAY [1..20] OF CHAR;
   VAR Names :  TEXT;
       Pres, Veep :  String;

     .
     .
     .

   READ(Names, Pres, Veep);
   ```

   This program fragment declares and reads the file Names, which contains the following characters:

   ```
   John F. Kennedy      Lyndon B. Johnson    Lyndon B. Johnson    <EOLN>
   Hubert H. Humphrey   <EOLN>
   Richard M. Nixon     Spiro T. Agnew       <EOLN>
   ```

   The first call to the READ procedure sets Pres equal to the 20-character string 'John F. Kennedy ' and Veep equal to 'Lyndon B. Johnson '. The second call to the procedure assigns the value 'Lyndon B. Johnson ' to Pres and 'Hubert H. Humphrey ' to Veep. A third call to READ reads in the last two names in the file.
   ```

3.
```
TYPE Color = (Red, Fire_Engine_Green, Blue, Black):
VAR Light : Color;
BEGIN
  READ(Light):
```

In this example, if R is input, the variable color takes on the value of Red. However, if Redx were entered an error would occur.

If just Bl were input an error would occur since Bl is not unique. However, Blu is sufficiently unique to be assigned to the constant Blue.

7.2.3 The READLN Procedure

The READLN procedure reads lines of data from a text file.

Format

```
READLN ( file-variable ||,variable-name ||,variable-name... || ||) ;
```

or

```
READLN ||( variable-name ||,variable-name... || ) || ;
```

file-variable

Specifies the name of the text file to be read. If you do not specify a file variable, PASCAL uses INPUT by default.

variable-name

Specifies the variable into which a value will be read. If you do not specify any variable names, READLN skips a line in the specified file.

The READLN procedure reads values from a text file. After reading values for all the listed variables, the READLN procedure skips over any characters remaining on the current line and positions the file at the beginning of the next line. All the values need not be on a single line; READLN continues until values have been assigned to all the specified variables, even if this process results in the reading of several lines of the input file. READLN performs the following sequence:

```
READ (file-variable, variable-name...);
READLN (file-variable);
```

EOLN() is TRUE only if the new line is empty.

You can use the READLN procedure to read integers, real numbers, characters, strings, or constants of enumerated types. The values in the file must be separated as for the READ procedure.

If EOLN() is TRUE when you call READLN, the first value read is the first value in the next line unless you are reading a character. If you are reading a character, the first value read is a space.

Examples

```
TYPE String = PACKED ARRAY [1..20] OF CHAR;
VAR Names : TEXT;
    Pres, Veep : String;

  .
  .
  .

READLN(Names, Pres, Veep);
```

This program fragment declares and reads the file Names, which contains the following characters:

```
John F. Kennedy      Lyndon B. Johnson    Lyndon B. Johnson    <EOLN>
Hubert H. Humphrey   <EOLN>
Richard M. Nixon     Spiro T. Agnew       <EOLN>
<EOLN>
<EOF>
```

The READLN procedure reads the values 'John F. Kennedy ' for Pres and 'Lyndon B. Johnson ' for Veep. It then skips to the next line, ignoring the remaining characters on the first line. Subsequent execution of the procedure assigns the value 'Hubert H. Humphrey ' to Pres and 'Richard M. Nixon ' to Veep, then skips to the next line setting EOLN equal to TRUE. A third call to the procedure sets EOF(Names) to TRUE and generates an error.

7.2.4 The RESET Procedure

The RESET procedure readies a file for reading.

Format

```
RESET (file-variable);
```

file-variable

Specifies the file to be read.

If the file is not already open, RESET opens it using the defaults listed in Tables 7-1 and 7-2.

After execution of RESET, the file is positioned at the first component, and the file buffer variable contains the value of this component. The arrow in Figure 7-2 shows the file position after RESET. If the file is empty, EOF() is TRUE; EOF() is FALSE otherwise. If the file does not exist, RESET does not create it, but returns an error at run time.

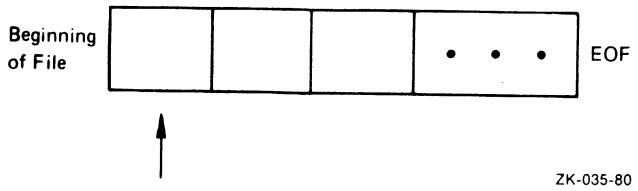


Figure 7-2 File Position after RESET

You must call RESET before reading any file except the predeclared file INPUT. The RESET procedure removes the end-of-file marker from the file INPUT. If you call RESET for the predeclared file OUTPUT, a compile-time error occurs.

Examples

1.
```
OPEN (Phones,'Phones.Dat',ACCESS_METHOD := Direct);
RESET (Phones);
```

These statements open the file variable Phones for direct access on input. After execution of the OPEN and RESET procedures, you can use the FIND procedure for direct access to the components of the file Phones.

2.
```
RESET (Weights);
```

If the file variable Weights is already open, this statement enables reading and sets Weights^ to the first file component. If the file is not open, this statement causes the system to search for the file designated by the logical name Weights. If no such logical name is assigned, the system searches for the file Weights.Dat on the user's default node, device, and directory. If Weights.DAT exists, the system opens it for sequential access. If Weights.DAT does not exist, an error occurs at run time.

Output Procedures

7.3 OUTPUT PROCEDURES

This section describes the following output procedures:

- LINELIMIT
- PAGE
- PUT
- REWRITE
- WRITE
- WRITELN

LINELIMIT

7.3.1 The LINELIMIT Procedure

The LINELIMIT procedure terminates execution of the program after a specified number of lines have been written into a text file.

Format

```
LINELIMIT (file-variable, n);
```

file-variable

Specifies the text file to which this limit applies.

n

Represents a positive integer expression indicating the number of lines that can be written to the file before execution terminates.

When the VAX-11 PASCAL compiler is installed, the system manager can specify a default line limit for all output text files. This default limit is optional and will vary from installation to installation. You can override your installation's default by calling LINELIMIT with a smaller or larger value for n.

You can specify a line limit for a file even if your system manager did not install the compiler with a default line limit. Call the LINELIMIT procedure and indicate the name of the file and the required limit.

After the number of lines output to the file has reached the line limit, program execution terminates.

Examples

```
LINELIMIT (Debts,100);
```

Execution of the program terminates after 100 lines have been written into the text file Debts.

PAGE

7.3.2 The PAGE Procedure

The PAGE procedure skips to the next page of a text file.

Format

```
PAGE (file-variable);
```

file-variable

Specifies a text file.

Execution of the PAGE procedure involves the system performing a WRITELN, to clear the record buffer, and then advancing output to the next page of the specified text file. The next line written to the file begins on the second line of a new page. You can use this procedure only on text files. If you specify a file of any other file type, PASCAL issues an error message.

The value of the page eject record that is output to the file depends on the carriage control format for that file. When CARRIAGE is enabled, the page eject record is equivalent to the carriage control character '1'. When LIST or NOCARRIAGE is enabled, the page eject record is a form-feed character.

Examples

1.
```
PAGE (Userguide);
```

   This example causes a page eject record to be written in the text file Userguide.

2.
```
PAGE (OUTPUT);
```

   This example calls the PAGE procedure for the predeclared file OUTPUT. As a result of this procedure, a page eject record is output at the terminal (in interactive mode) or in the batch log file (in batch mode).

PUT

7.3.3 The PUT Procedure

The PUT procedure appends a new component to the end of a file.

Format

```
PUT (file-variable);
```

file-variable

Specifies the input file.

Before executing the PUT procedure, you must execute the REWRITE procedure. REWRITE clears the file and sets EOF() to TRUE, preparing the file for output. If EOF() is FALSE, the PUT procedure fails; a run-time error occurs and program execution is terminated.

The PUT procedure writes the value of the file buffer variable at the end of the specified file. After execution of the PUT procedure, the value of the file buffer variable becomes undefined. EOF() remains TRUE.

Examples

```
PROGRAM Bookfile (INPUT,OUTPUT,Books);

TYPE String = PACKED ARRAY[1..40]OF CHAR;
     Bookrec = RECORD
               Author : String;
               Title : String
               END;
VAR   Newbook : Bookrec;
     Books : FILE OF Bookrec;
     N : INTEGER;

BEGIN
  REWRITE (Books);
  FOR N := 1 TO 10 DO BEGIN
     WITH Newbook DO BEGIN
        WRITE ('Title:');
        READLN (Title);
        WRITE ('Author:');
        READLN (Author)
        END;
     Books^ := Newbook;
     PUT (Books)
     END
END.
```

This program writes the first 10 records into the file Books. The records are input from the terminal to the record variable Newbook. They consist of two 40-character strings denoting a book's author and title. The FOR loop accepts 10 values for Newbook, assigning each new record to the file buffer variable Books^. The PUT statement writes the value of Books^ into the file for each of the 10 records input.

REWRITE

7.3.4 The REWRITE Procedure

The REWRITE procedure readies a file for output.

Format

```
REWRITE (file-variable);
```

file-variable

Specifies the file to be enabled for output.

If the file variable has not been opened, REWRITE creates and opens it using the defaults listed in Tables 7-1 and 7-2.

You must call REWRITE before writing any file except the predeclared file OUTPUT. If you call REWRITE for the predeclared files INPUT or OUTPUT, a compile-time error occurs.

The REWRITE procedure truncates the file to length zero and sets EOF() to TRUE. You can then write new components into the file with the PUT, WRITE, or WRITELN procedure (WRITELN is defined only for text files). After the file is open, successive calls to REWRITE truncate the existing file to length zero; they do not create new versions of the file.

To update an existing file, you must copy its contents to another file, specifying new values for the components that you need to update.

Examples

1.
```
REWRITE (Storms);
```

   If the file variable Storms is already open, this statement enables writing, clears the file of old data, and sets the file position to the beginning of the file. If Storms is not open, a new version is created with the same defaults as for the OPEN procedure (Section 7.1.5).

2.
```
OPEN (Ratings, '[Insurance]CARS.DAT' HISTORY := OLD, RECORD TYPE
:= FIXED); REWRITE (Ratings);
```

   The OPEN procedure opens the file variable Ratings, which is associated with the VAX/VMS file CARS.DAT in directory [Insurance]. The REWRITE procedure discards the current contents of the file Ratings and sets the file position at the beginning of the file. After execution of this statement, EOF(Ratings) is TRUE.

WRITE

7.3.5 The WRITE Procedure

The WRITE procedure outputs data to a file.

Format

```
WRITE ([[file-variable,]] print-list );
```

file-variable

Specifies the file to be written. If you omit the file variable, PASCAL uses OUTPUT by default.

print-list

Specifies the values to be output, separated by commas. The print list can contain constants, variables, and expressions. For nontext files, the items in the print list must be assignment compatible with the file component type.

By definition, a WRITE to a nontext file performs an assignment to the file buffer variable and a PUT for each expression. Thus, the procedure call

```
WRITE (file-variable, expression);
```

is equivalent to

```
file-variable^ := expression;
PUT (file-variable);
```

For text files, the WRITE procedure converts each item in the print list to a sequence of characters. It repeats the assignment and PUT process until all the items in the list have been written in the file.

The print list can specify expressions, constants, variable names, array elements, strings, and record fields, with values of any scalar type. Each value is output with a minimum field width, as specified in Table 7-3.

Table 7-3
Default Values for Field Width

Type of Variable Printed	Number of Characters
Integer	10
Real	16
Double	24
Character	1
Enumerated	Size of identifier + 1 up to 32
BOOLEAN	6

You can override these defaults for a particular value by specifying a field width in the print list. The field width specifies the minimum number of characters to be output for the value. The following is the format of the field width specification:

```
expression : minimum : fraction
```

Both minimum and fraction represent positive integer expressions. The minimum indicates the minimum number of characters to be output for the value. The fraction, which is permitted only for real values, indicates the number of digits to the right of the decimal point.

By default, PASCAL prints real numbers in floating-point format. Each real number is preceded by a blank or a minus sign and the rightmost digit is rounded. For example:

```
WRITE (Shoesize);
```

If the value of Shoesize is 12.5, this statement produces the following output:

```
1.250000000E+01
```

To print the value in decimal format, you must specify a field width as in this example:

```
WRITE (Shoesize:5:1);
```

The first integer indicates that a minimum of five characters will be output. The minimum includes the leading blank, minus sign, and the decimal point. The second integer specifies one digit to the right of the decimal point. This statement results in the following output:

```
12.5
```

If the print field is wider than necessary, PASCAL prints the value with leading blanks.

If you try to print an enumerated type, a real or an integer value in a field that is too narrow, PASCAL truncates on the left. The PASCAL compiler does not check for the uniqueness of the truncated identifier.

For an expression of an enumerated type, PASCAL prints the constant identifier denoting the variable's value. For example:

```
VAR Color :  (Blue, Yellow, Black, Fire_Engine_Green);
  .
  .
  .
WRITE ('My favorite color is ',Color:15);
```

When the value of Color is Yellow, the following is printed:

```
My favorite color is              YELLOW
```

When the value of Color is Fire_Engine_Green the following appears:

```
My favorite color is   FIRE_ENGINE_GRE
```

Since the field width specified is not wide enough to print all 17 characters in the identifier, PASCAL truncates the last two characters. Note that constants of enumerated types are printed in all uppercase characters.

If you open the predeclared file OUTPUT with the default carriage control option LIST, VAX-11 PASCAL allows you to use the WRITE procedure to prompt for input at the terminal. Each time you read from INPUT, the system checks for any output in the terminal record buffer. If the buffer contains any characters, the system prints them at the terminal, but suppresses the carriage return at the end of the line. The output text appears as a prompt, and you can type your input on the same line. For example:

```
WRITE ('Name three presidents:');
READ (Pres1, Pres2, Pres3);
```

When it executes the READ procedure, the system finds the output string waiting to be printed. It prints the prompt at the terminal, leaving the carriage just after the colon (:). You can then begin typing input on the same line as the prompt.

Prompting works only for the predeclared files INPUT and OUTPUT. For any other files, no output is written until you fill the output record buffer or start a new line using WRITELN.

Examples

1.
```
TYPE String = PACKED ARRAY [1..20] OF CHAR;
VAR Names :  FILE OF String;
    Pres :  String;
  .
  .
  .
WRITE (Names, 'Millard Fillmore    ', Pres);
```

This example writes two components in the file Names. The first is the 20-character string constant 'Millard Fillmore '. The second is the string variable Pres.

2.
```
WRITE (Num1:5:1,' and',Num2:5:1,' sum to',(Num1 + Num2):6:1);
```

If you specify an expression, PASCAL prints its value. For example, if Num1 equals 71.1 and Num2 equals 29.9, this statement prints:

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

Note that the chosen field width causes each of the real numbers is preceded by a space.

3.
```
VAR Rainamts : FILE OF REAL;
    Avg_Rain,Max_Rain,Min_Rain : REAL;
    .
    .
    .
WRITE (Rain_Amts, Avg_Rain, Min_Rain,0.312, Max_Rain);
```

The file Rain_Amts contains real numbers indicating amounts of rain_Fall. The WRITE procedure writes the values of the variables Avg_Rain and Min_Rain into the file followed by the real constant 0.312 and the value of the variable Max_Rain.

7.3.5.1 **Printing Hexadecimal Values Using WRITE** - The following format of the WRITE procedure is used to print hexadecimal values.

Format

WRITE (expression:field-width HEX [[expression:field-width HEX...]]);

expression

Specifies the value to be output in hexadecimal notation. Arbitrary items (including pointers) may be written in hexadecimal to text files.

field-width

Specifies the length of the print field.

If the field width specified is less than eight characters, and the output is greater than the field width, the value being printed is truncated on the left. If the field width is greater than eight characters, and the output is less than the field width, the field is padded with blanks on the left.

Examples

WRITE (Payroll:10 HEX)

The variable Payroll is printed using up to 10 hexadecimal characters.

7.3.5.2 **Printing Octal Values Using WRITE** - The following format of the WRITE procedure is used to print octal values.

Format

WRITE (expression:field-width OCT);

expression

Specifies the value to be output in octal notation. Arbitrary items (including pointers) may be written in hexadecimal to text files.

field-width

Specifies the length of the print field.

If the field width specified is less than 11 characters, and the output is greater than the field width, the value being printed is truncated on the left. If the field width is greater than 11 characters, and the output is less than the field width, the field is padded with blanks on the left.

Examples

WRITE (Social_Security:14 OCT)

The variable Social_Security is printed.

WRITELN

7.3.6 The WRITELN Procedure

The WRITELN procedure writes a line of data in a text file.

Format

```
WRITELN ( file-variable, [[print-list]] );
```

or

```
WRITELN [[( print-list )]] ;
```

file-variable

Specifies the text file to be written. If you omit the file variable, PASCAL uses OUTPUT by default.

print-list

Specifies the values to be output, separated by commas.

The print list can specify expressions, constants, variable names, array elements, or record fields, with values of any scalar type. Output of strings is also permitted. Each value is output with a minimum field width, as described in Section 7.3.5.

The WRITELN procedure writes the specified values into the text file, then starts a new line. For example:

```
WRITELN (Userguide, 'This manual describes how you interact');
```

As a result of this statement, the system writes the string in the text file Userguide and skips to the next line.

Hexadecimal and octal values can be printed using the same hexadecimal or octal format as the WRITE procedure.

When you open a text file, you can specify the CARRIAGE or FORTRAN option for carriage-control format. If you select CARRIAGE format, the first character of each output line is treated as a carriage-control character when output is directed to carriage-control devices such as the terminal or line printer. If output is not directed to a carriage control device, the carriage control character is written into the file and will be read back when you open the file for input. Table 7-4 summarizes the carriage control characters and their effects.

For carriage control purposes, any characters other than those listed in the table are ignored.

The carriage control character must be the first item in an output text line. For example, if the text file TREE is open with the CARRIAGE option, you can use the following statement:

```
WRITELN (Tree,' ',String1, String2);
```

The first item in the print list is a space character. The space indicates that the values of String1 and String2 are to be printed beginning on a new line when the file is output to a terminal, line printer, or similar carriage control device.

Table 7-4
Carriage Control Characters

Character	Meaning
'+'	Overprinting: starts output at the beginning of the current line
space	Single spacing: starts output at the beginning of the next line
'0'	Double spacing: skips a line before starting output
'1'	Paging: starts output at the top of a new page
'$'	Prompting: starts output at the beginning of the next line, and suppresses carriage return at the end of the line

If you select CARRIAGE format, you can use the dollar sign ($) character to initiate prompting at the terminal. Open the predeclared file OUTPUT, specifying only CARRIAGE. Then, use statements like the following to prompt for input at the terminal:

```
WRITELN ('$How many inches of rain last night?');
```

This statement prints the text at the terminal and suppresses the carriage return. The answer can be typed at the end of the line on which the prompt appears.

If you specify CARRIAGE but use an invalid carriage control character, the first character in the line is ignored. The output appears with the first character truncated.

Examples

1. `WRITELN (Class[1]:2,' is the grade for this student.');`

   This example writes an element of the character array Class to the file OUTPUT. The value is written with a minimum field width of 2.

2. `WRITELN;`

   If you specify WRITELN without a file variable or print list, it ends the printing of the current line on the standard output device (usually the terminal).

3.
```
TYPE String : PACKED ARRAY [1..40] OF CHAR;
VAR  Newhires : TEXT;
     N : INTEGER:
     Newrec : RECORD
              Id : INTEGER;
              Name : String;
              Address : String;
              Salary : String
              END;
.
.
.
OPEN (Newhires, CARRIAGE_CONTROL:= Carriage);
WITH Newrec Do BEGIN
   WRITELN (Newhires, '1New hire # ',ID:1,' is ' ,Name);
   WRITELN (Newhires, ' ', Name, ' lives at:');
   WRITELN (Newhires,' ');
   WRITELN (Newhires,' ', Address)
   END;
```

This example writes four lines in the text file Newhires. The output starts at the top of a new page, and fits the following format:

```
New hire # 73 is Irving Washington
Irving Washington lives at:

22 Chestnut St, Seattle, Wash.
```

7.4 TERMINAL I/O

In VAX-11 PASCAL the file pointer of the text file INPUT is not physically advanced to the next character or data before the first read operation or after the end-of-line marker is passed. The file pointer is held in a suspended state until the next character of data is actually required. The actual reading of the data occurs when one of the following is executed:

```
GET(INPUT)

READ(INPUT)

READLN(INPUT)

EOLN(INPUT)

EOF(INPUT)

When the file buffer INPUT^ is accessed
```

This concept is called lazy lookahead. When programming a terminal, the placement of the first prompt for input must take into account lazy lookahead.

You place the first input prompt before any requests for EOF, EOLN or a request for input. The placement of the prompt before an EOF, EOLN or read supplies the first character of data.

The following example shows a prompt for input that is placed before the EOLN check.

```
VAR Ch: CHAR;
BEGIN
WRITE ( 'ENTER A CHARACTER OR A EMPTY LINE');
  BEGIN
  WHILE NOT EOLN DO
      READ (Ch);
  READLN;
  WRITE('ENTER A CHARACTER OR AN EMPTY LINE');
 END;
END.
```

APPENDIX A

ASCII CHARACTER SET

Table A-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 A-1 is the same as the ordinal value (as returned by the PASCAL ORD function) of the associated character in the type CHAR.

Table A-1
ASCII Character Set

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

(continued on next page)

Table A-1 (Cont.)
ASCII Character Set

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
64	@	At sign	96	`	Grave accent
65	A	Upper case A	97	a	Lower case a
66	B	Upper case B	98	b	Lower case b
67	C	Upper case C	99	c	Lower case c
68	D	Upper case D	100	d	Lower case d
69	E	Upper case E	101	e	Lower case e
70	F	Upper case F	102	f	Lower case f
71	G	Upper case G	103	g	Lower case g
72	H	Upper case H	104	h	Lower case h
73	I	Upper case I	105	i	Lower case i
74	J	Upper case J	106	j	Lower case j
75	K	Upper case K	107	k	Lower case k
76	L	Upper case L	108	l	Lower case l
77	M	Upper case M	109	m	Lower case m
78	N	Upper case N	110	n	Lower case n
79	O	Upper case O	111	o	Lower case o
80	P	Upper case P	112	p	Lower case p
81	Q	Upper case Q	113	q	Lower case q
82	R	Upper case R	114	r	Lower case r
83	S	Upper case S	115	s	Lower case s
84	T	Upper case T	116	t	Lower case t
85	U	Upper case U	117	u	Lower case u
86	V	Upper case V	118	v	Lower case v
87	W	Upper case W	119	w	Lower case w
88	X	Upper case X	120	x	Lower case x
89	Y	Upper case Y	121	y	Lower case y
90	Z	Upper case Z	122	z	Lower case z
91	[	Left square bracket	123	{	Left brace
92	\	Back slash	124	|	Vertical line
93	]	Right square bracket	125	}	Right brace
94	^ or ↑	Circumflex or up arrow	126	~	Tilde
95	← or _	Back arrow or underscore	127	DEL	Delete

APPENDIX B

SYNTAX SUMMARY

This appendix summarizes the syntax of VAX-11 PASCAL in the Backus-Naur Form (BNF) and presents syntax diagrams in the format commonly used for PASCAL.

B.1 BACKUS-NAUR FORM

In the BNF, each element of the language is defined recursively in terms of simpler elements. The element being defined is written to the left of the symbol ::= and its definition is written to the right of that symbol.

The BNF uses a group of meta-symbols that differ from the conventions used in the rest of this manual and are not part of the PASCAL language. Table B-1 summarizes the meta-symbols used in the BNF.

Table B-1
BNF Meta-Symbols

Symbol	Meaning
::=	Separates the element being defined from its definition.
< >	Enclose a definable language element.
[ ]	Enclose an optional element.
|	Means "or"; separates possible elements.
...	Encloses elements that can be repeated one or more times, but need not be present.

The remainder of this section lists VAX-11 PASCAL in BNF (except for the BNF for I/O statements).

SYNTAX SUMMARY

<compilation unit> ::= <program> | <module>

<module> ::= <module heading> <declaration part>
<procedure and function declaration part> END.

<program> ::= <program heading> <block> .

<module heading> ::= MODULE <identifier> ; |
MODULE <identifier> (<program parameters>) ;

<program heading> ::= PROGRAM <identifier> ; |
PROGRAM <identifier> (<program parameters>) ;

<program parameters> ::= <external file identifier>
{ , <external file identifier> }

<external file identifier> ::= <identifier>

<identifier> ::= <letter> { <alphanumeric> }

<alphanumeric> ::= <letter> | <digit> | _ | $

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z |

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<block> ::= <declaration part> <value initialization part>
<procedure and function declaration part> <statement part>

<declaration part> ::= <label declaration part> <constant definition part>
<type definition part> <variable declaration part>

<label declaration part> ::= <empty> | LABEL <label> { , <label> } ;

<label> ::= <unsigned integer>

<constant definition part> ::= <empty> |
CONST <constant definition> { ; <constant definition> } ;

<constant definition> ::= <identifier> = <constant>

<constant> ::= <unsigned number> | <sign> <unsigned number> |
<constant identifier> | <sign> <constant identifier> |
<string> | NIL

<unsigned number> ::= <unsigned integer> | <unsigned real>

<unsigned integer> ::= <digit sequence> | <radix integer>

<radix integer> ::= %C <octal integer> | %H <hex integer>

<octal integer> ::= <letter o> <octal digit sequence>

<letter o> ::= O | o

<octal digit sequence> ::= <octal digit> { <octal digit> }

<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<hex integer> ::= <letter x> <hex digit sequence>

<letter x> ::= X | x

<hex digit sequence> ::= <hex digit> | <hex digit> |

<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f

<unsigned real> ::= <unsigned single> | <unsigned double>

<unsigned single> ::= <digit sequence> . <digit sequence> |
<digit sequence> . <digit sequence> E <scale factor> |
<digit sequence> . <digit sequence> e <scale factor> |
<digit sequence> E <scale factor> |
<digit sequence> e <scale factor>

<unsigned double> ::= <digit sequence> . <digit sequence> D <scale factor>
<digit sequence> . <digit sequence> d <scale factor> |
<digit sequence> D <scale factor> |
<digit sequence> d <scale factor>

<digit sequence> ::= <digit> | <digit> |

<scale factor> ::= <digit sequence> | <sign> <digit sequence>

<sign> ::= + | -

<constant identifier> ::= <identifier>

<string> ::= ' <character> | <character> |

<character> ::= <any ASCII character except '> | ''

<type definition part> ::= <empty> |
TYPE <type definition> | ; <type definition> | ;

<type definition> ::= <identifier> = <type>

<type> ::= <simple type> | <structured type> | <pointer type>

<simple type> ::= <scalar type> | <subrange type> | <type identifier>

<scalar type> ::= (<identifier> | , <identifier> |)

<subrange type> ::= <constant> .. <constant>

<type identifier> ::= <identifier>

<structured type> ::= <unpacked structured type> |
PACKED <unpacked structured type>

<unpacked structured type> ::= <array type> | <record type> |
<set type> | <file type>

<array type> ::= ARRAY [<index type> | , <index type> |] OF
<component type>

<index type> ::= <simple type>

<component type> ::= <type>

<record type> ::= RECORD <field list> END

<field list> ::= <fixed part> | <fixed part> ; <variant part> |
<variant part>

<fixed part> ::= <record section> | ; <record section> |

<record section> ::= <empty> |
<field identifier> { , <field identifier> } : <type>

<variant part> ::= CASE <tag field> <type identifier> OF
<variant> { ; <variant> }

<tag field> ::= <field identifier> : | <empty>

<variant> ::= <case label list> : (<field list>) | <empty>

<case label list> ::= <case label> { , <case label> }

<case label> ::= <constant>

<set type> ::= SET OF <base type>

<base type> ::= <simple type>

<file type> ::= FILE OF <type>

<pointer type> ::= ^ <type identifier>

<variable declaration part> ::= <empty> |
VAR <variable declaration> { ; <variable declaration> } ;

<variable declaration> ::= <identifier> { , <identifier> } : <type>

<value initialization part> ::= <empty> |
VALUE <value initialization> { ; <value initialization> } ;

<value initialization> ::= <identifier> := <value>

<value> ::= <constant> | <set constant> | <constructor>

<set constant> ::= [<constant element list>]

<constant element list> ::= <empty> |
<constant element> { , <constant element> }

<constant element> ::= <constant> | <constant> .. <constant>

<constructor> ::= <optional type> (<value element> { , <value element> })

<optional type> ::= <empty> | <type identifier>

<value element> ::= <value> | <repetition factor> OF <value>

<repetition factor> ::= <unsigned integer> | <constant identifier>

<procedure and function declaration part> ::=
{ <procedure or function declaration> ; }

<procedure or function declaration> ::= <procedure declaration> |
<function declaration>

<procedure declaration> ::= <internal procedure declaration> |
<external procedure declaration> |
<forward procedure declaration>

<internal procedure declaration> ::= <procedure heading> <block>

<external procedure declaration> ::= <procedure heading> EXTERN |
<procedure heading> FORTRAN

<forward procedure declaration> ::= <procedure heading> FORWARD

<procedure heading> ::= PROCEDURE <identifier> ; |
PROCEDURE <identifier> (<formal parameter section>
| ; <formal parameter section> |) ;

<formal parameter section> ::= <extended parameter group> |
VAR <extended parameter group> |
%DESCR <extended parameter group> |
%STDESCR <extended parameter group> |
%IMMED <parameter group> |
FUNCTION <parameter group> |
%IMMED FUNCTION <parameter group> |
PROCEDURE <identifier> | , <identifier> | |
%IMMED PROCEDURE <identifier> | , <identifier> |

<extended parameter group> ::= <parameter group> |
<identifier> | , <identifier> | : <dynamic array type>

<dynamic array type> ::= ARRAY [<scalar type identifier>
| , <scalar type identifier> |] OF <type identifier> |
PACKED ARRAY [<scalar type identifier>
| , <scalar type identifier> |] OF <type identifier>

<scalar type identifier> ::= <identifier>

<parameter group> ::= <identifier> | , <identifier> | : <type identifier>

<function declaration> ::= <internal function declaration> |
<external function declaration> |
<forward function declaration>

<internal function declaration> ::= <function heading> <block> |
FUNCTION <identifier>;<block>

<external function declaration> ::= <function heading> EXTERN |
<function heading> FORTRAN

<forward function declaration> ::= <function heading> FORWARD

<function heading> ::= FUNCTION <identifier> : <result type> ; |
FUNCTION <identifier> (<formal parameter section>
| ; <formal parameter section> |) : <result type> ;

<result type> ::= <type identifier>

<statement part> ::= <compound statement>

<statement> ::= <unlabeled statement> | <label> : <unlabeled statement>

<unlabeled statement> ::= <simple statement> | <structured statement>

<simple statement> ::= <assignment statement> | <procedure statement> |
<go to statement> | <empty statement>

<assignment statement> ::= <variable> := <expression> |
<function identifier> := <expression>

<variable> ::= <entire variable> | <component variable> |
<referenced variable>

<entire variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

<component variable> ::= <indexed variable> | <field designator> |
<file buffer>

<indexed variable> ::= <array variable> [<expression>
{ , <expression> }]

<array variable> ::= <variable>

<field designator> ::= <record variable> . <field identifier>

<record variable> ::= <variable>

<field identifier> ::= <identifier>

<file buffer> ::= <file variable> ^

<file variable> ::= <variable>

<referenced variable> ::= <pointer variable> ^

<pointer variable> ::= <variable>

<expression> ::= <simple expression> | <simple expression>
<relational operator> <simple expression>

<relational operator> ::= = | <> | < | <= | >= | > | IN

<simple expression> ::= <term> | <sign> <term> |
<simple expression> <adding operator> <term>

<adding operator> ::= + | - | OR

<term> ::= <primary> | <term> <multiplying operator> <primary>

<multiplying operator> ::= * | / | DIV | MOD | AND

<primary> ::= <factor> | <primary> ** <factor>

<factor> ::= <variable> | <unsigned constant> | (<expression>) |
<function designator> | <set> | NOT <factor>

<unsigned constant> ::= <unsigned number> | <string> |
<constant identifier> | NIL

<function designator> ::= <function identifier> |
<function identifier> (<actual parameter>
{ , <actual parameter> })

<function identifier> ::= <identifier>

<set> ::= [<element list>]

<element list> ::= <element> { , <element> } | <empty>

<element> ::= <expression> | <expression> .. <expression>

<procedure statement> ::= <procedure identifier> |
<procedure identifier> (<actual parameter>
{ , <actual parameter> })

<procedure identifier> ::= <identifier>

<actual parameter> ::= <expression> | <variable > |
<procedure identifier> | <function identifier>

<go to statement> ::= GOTO <label>

<empty statement> ::= <empty>

<empty> ::=

<structured statement> ::= <compound statement> |
<conditional statement> | <repetitive statement> |
<with statement>

<compound statement> ::= BEGIN <statement> | ; <statement> | END

<conditional statement> ::= <if statement> | <case statement>

<if statement> ::= IF <expression> THEN <statement> |
IF <expression> THEN <statement> ELSE <statement>

<case statement> ::= CASE <expression> OF <case list element>
| ; <case list element> | <otherwise part> <end case>

<case list element> ::= <case label list> : <statement> | <empty>

<otherwise part> ::= <empty> |
OTHERWISE <statement>

<end case> ::= END | ; END

<repetitive statement> ::= <while statement> | <repeat statement> |
<for statement>

<while statement> ::= WHILE <expression> DO <statement>

<repeat statement> ::= REPEAT <statement> | ; <statement> |
UNTIL <expression>

<for statement> ::= FOR <control variable> := <for list> DO <statement>

<for list> ::= <initial value> TO <final value> |
<initial value> DOWNTO <final value>

<control variable> ::= <identifier>

<initial value> ::= <expression>

<final value> ::= <expression>

<with statement> ::= WITH <record variable list> DO <statement>

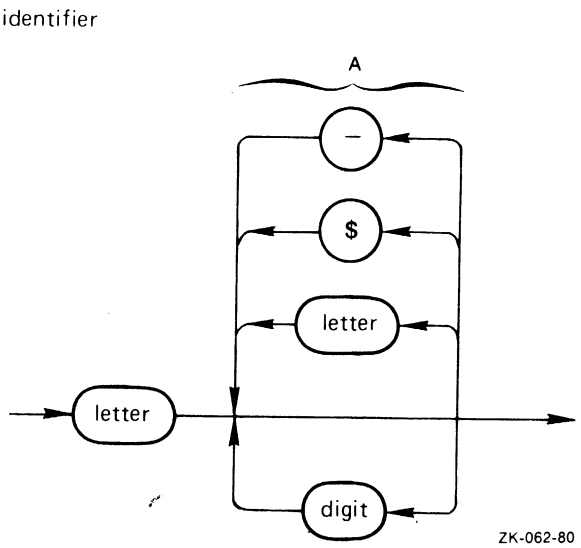
<record variable list> ::= <record variable> | , <record variable> |

B.2 SYNTAX DIAGRAMS

The following diagrams describe the syntax of the following items:

- Block
- Compilation unit
- Constant
- Declaration Section
- Dynamic array type
- Expression
- Factor
- Field list
- Hexadecimal digit
- Identifier
- Octal digit
- Parameter list
- Primary
- Procedure and function declaration section
- Simple expression
- Simple statement
- Simple type
- Statement
- Structured statement
- Term
- Type
- Unsigned constant
- Unsigned integer
- Unsigned number
- Value
- Variable

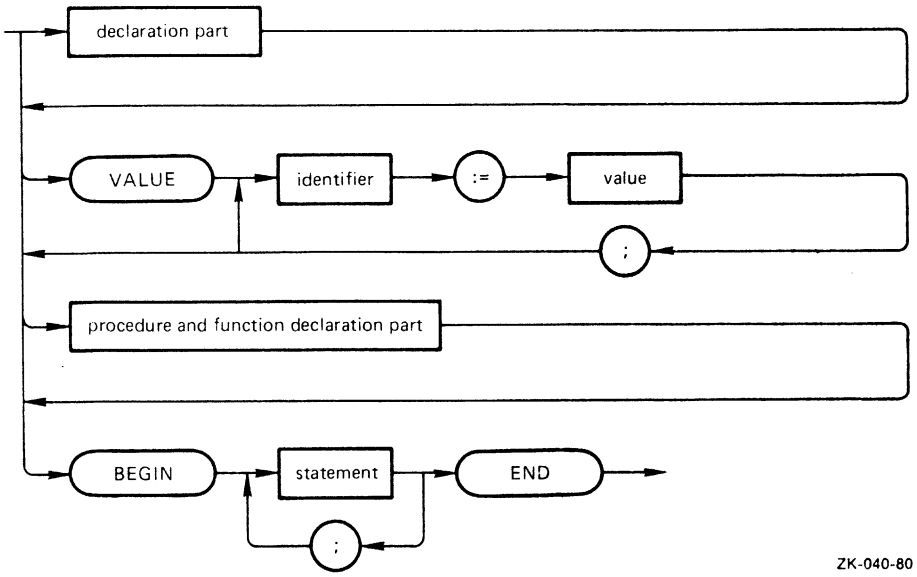
An example of how to use a diagram follows:



The first character of an identifier must be a letter. The next character is chosen from the section labeled A. In this example, the character can be a digit, a letter, a dollar sign ($), or a hyphen.

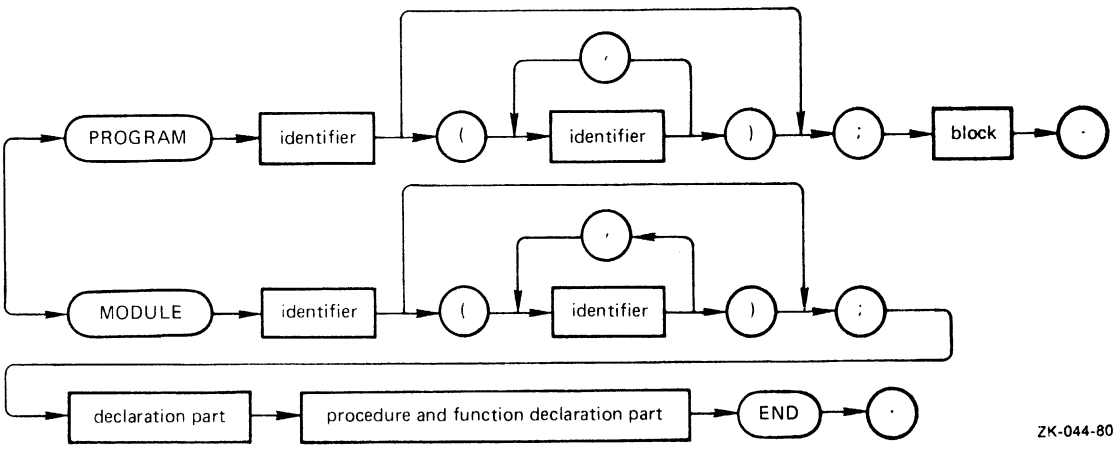
Section A is repeated until the identifier is defined.

block



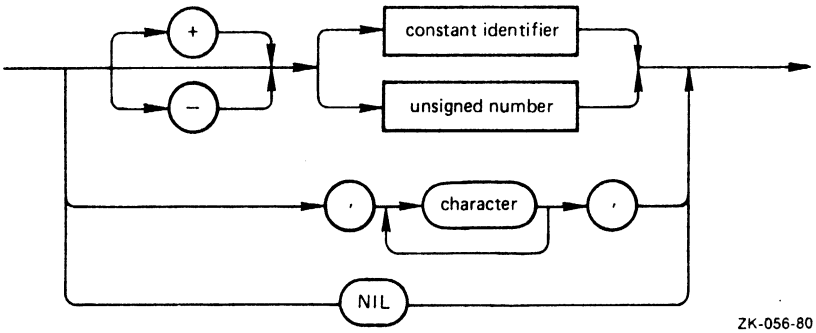
ZK-040-80

compilation unit



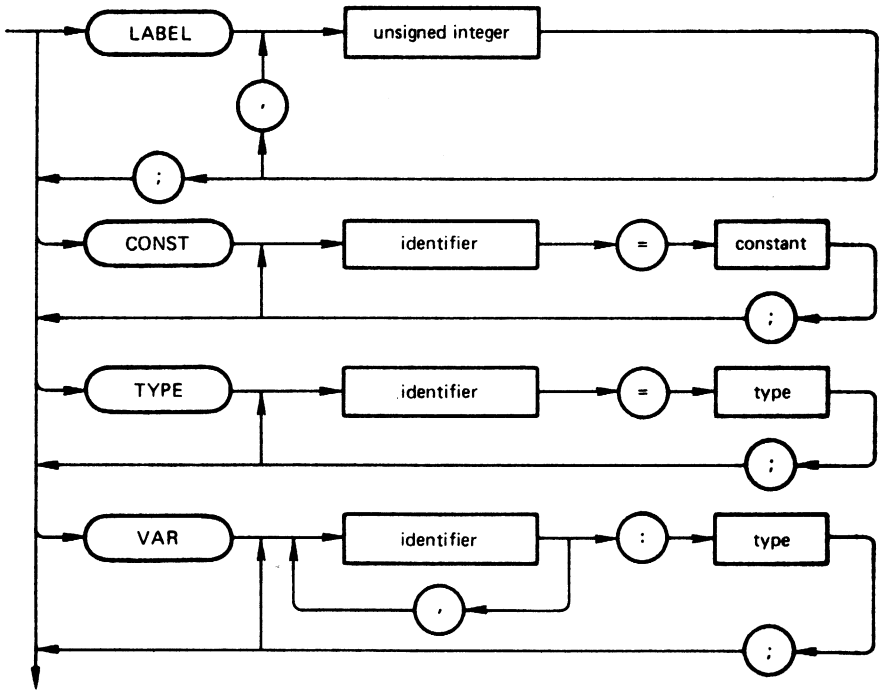
ZK-044-80

constant



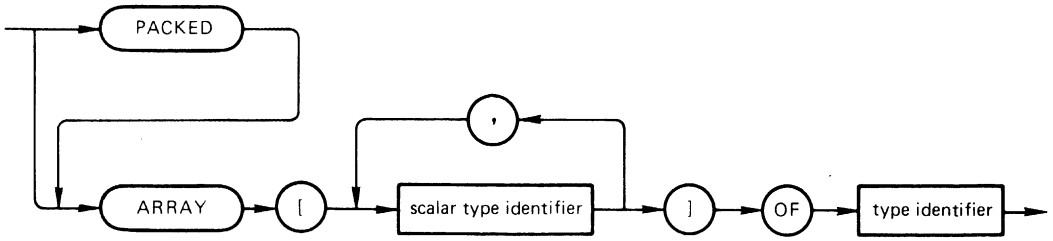
ZK-056-80

declaration section



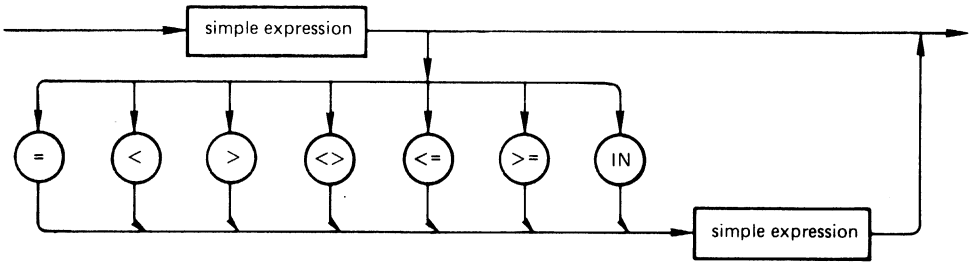
ZK-042-80

dynamic array type



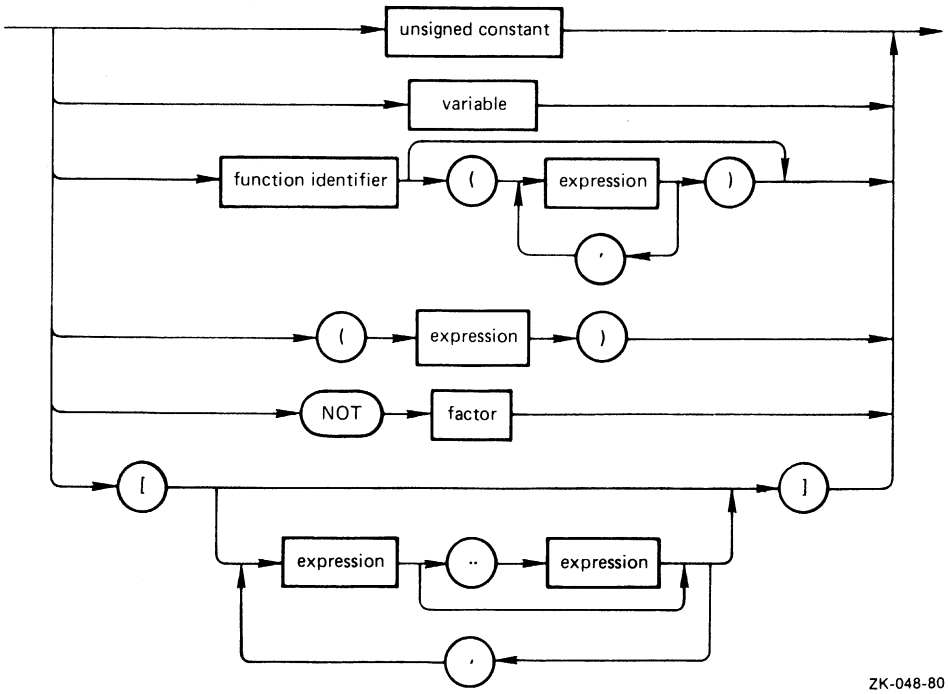
ZK-036-80

expression



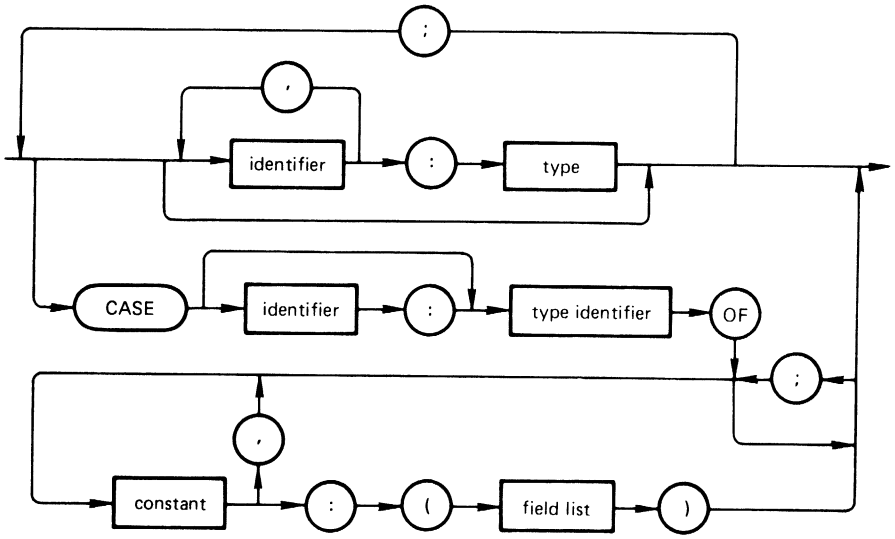
ZK-052-80

factor



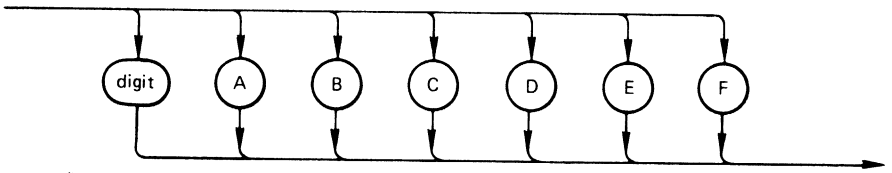
ZK-048-80

field list



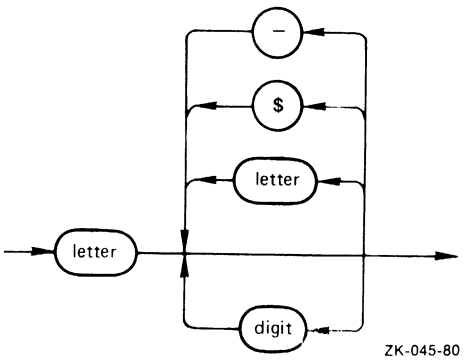
ZK-059-80

hexadecimal digit



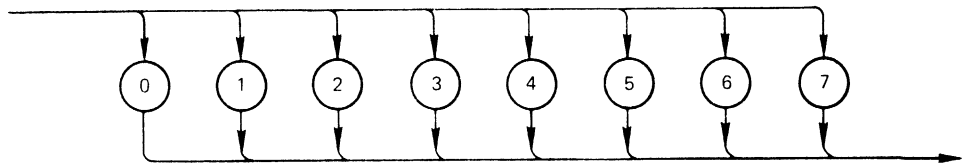
ZK-054-80

identifier



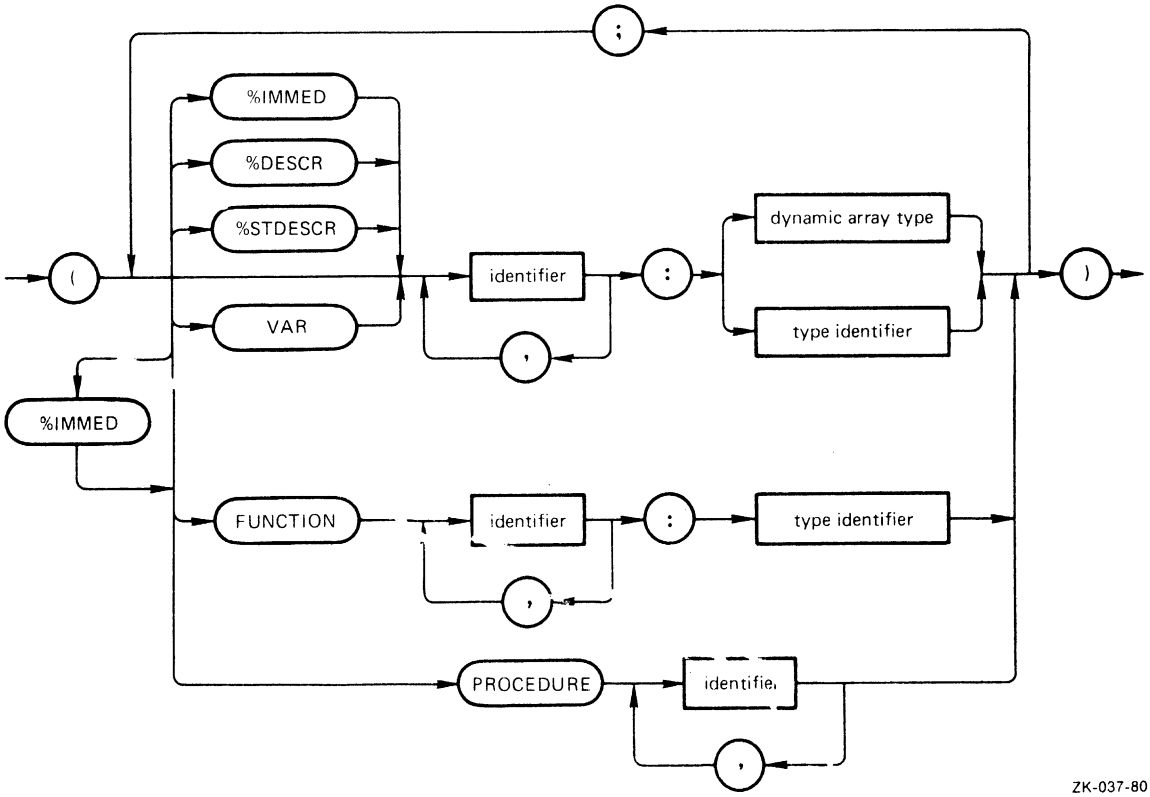
ZK-045-80

octal digit



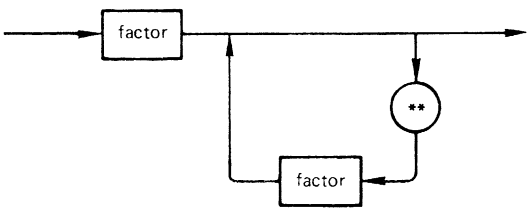
ZK-053-80

parameter list



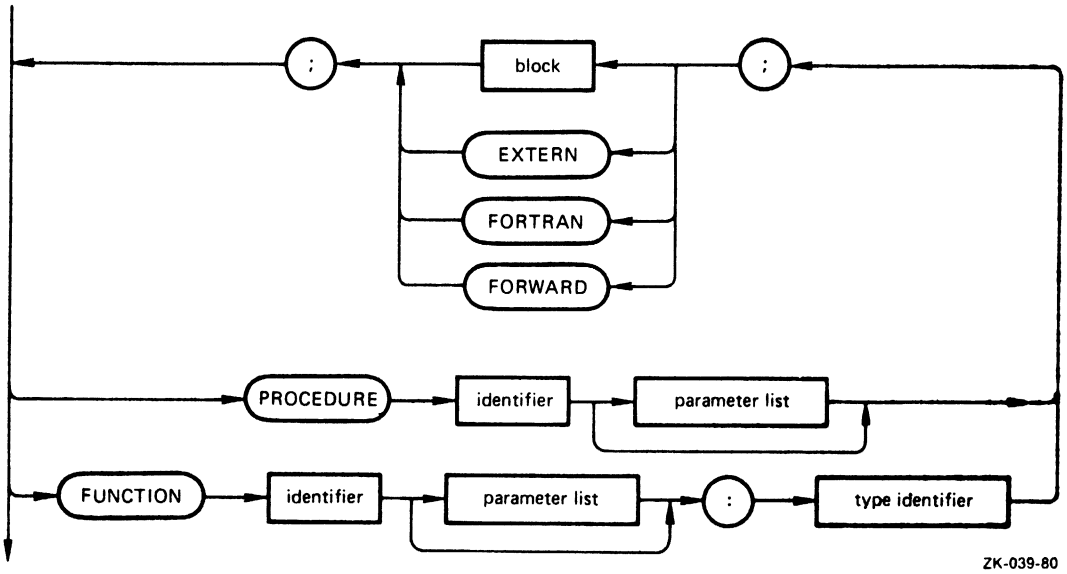
ZK-037-80

primary



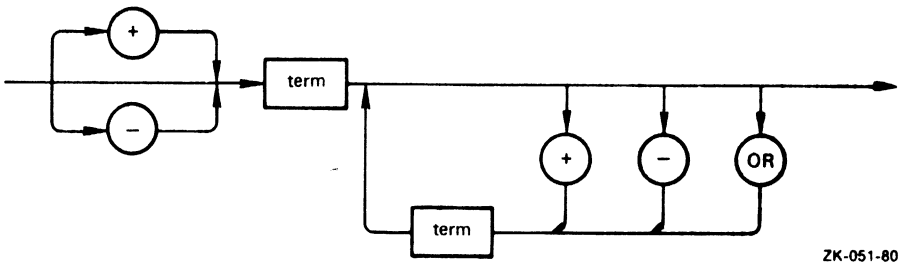
ZK-049-80

procedure and function declaration section



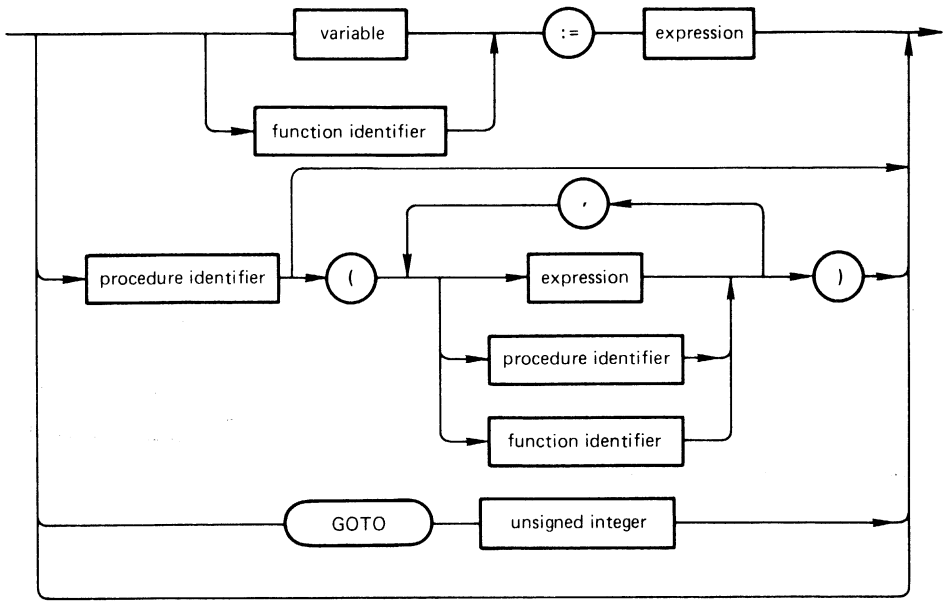
ZK-039-80

simple expression



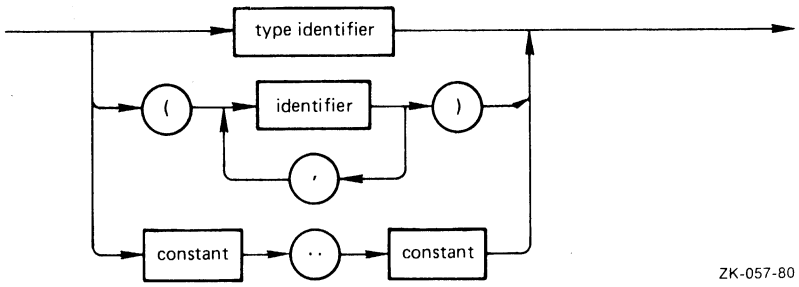
ZK-051-80

simple statement



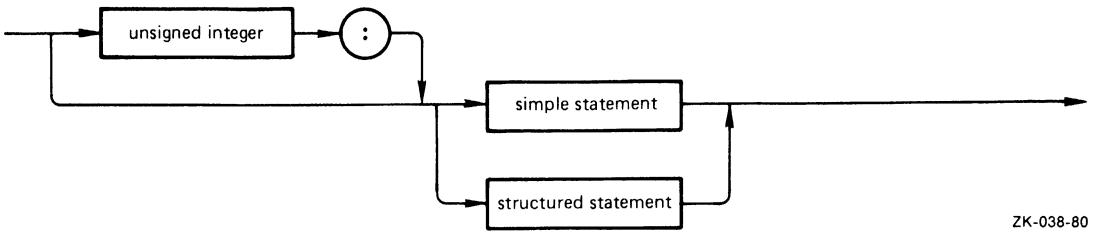
ZK-041-80

simple type



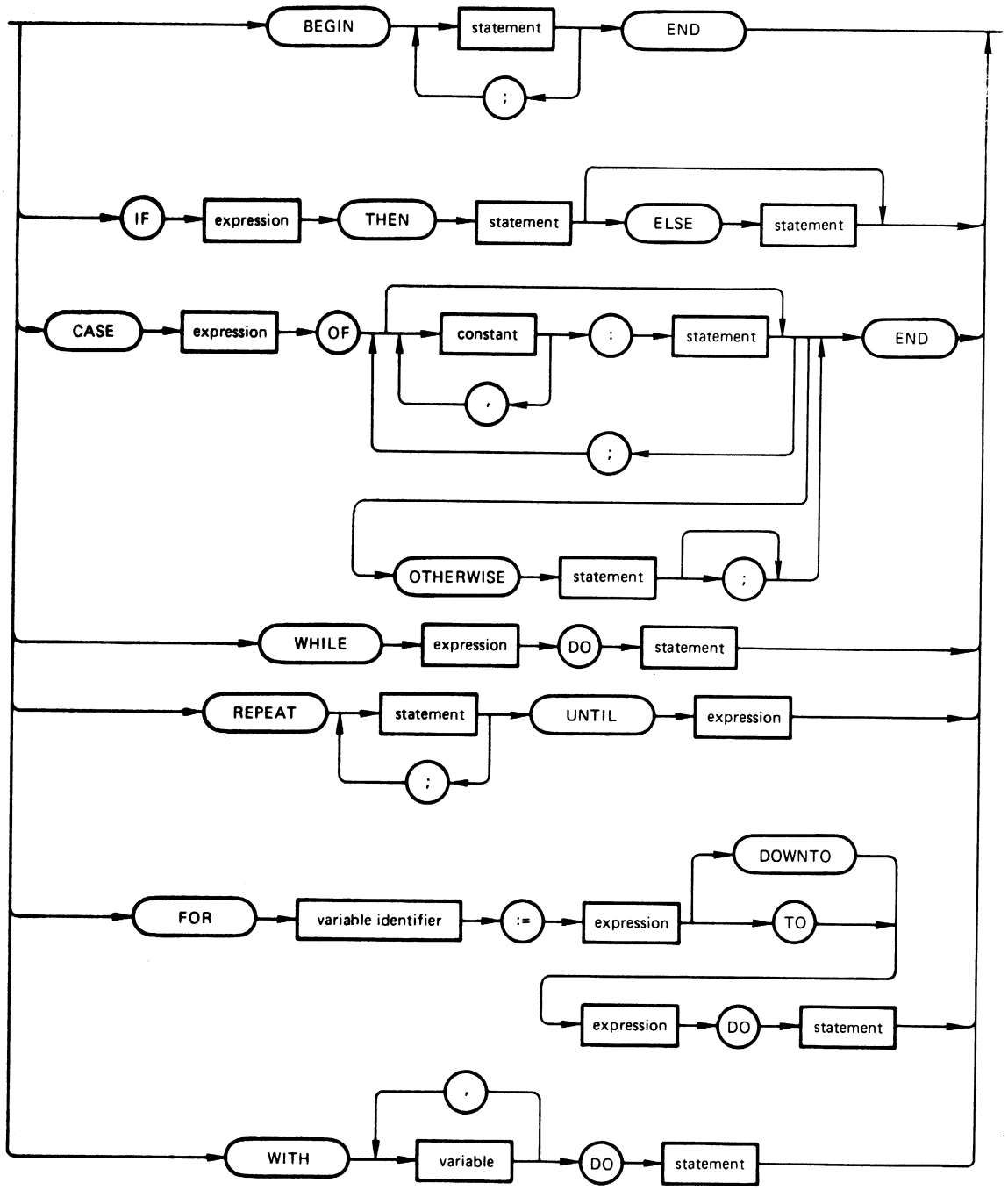
ZK-057-80

statement

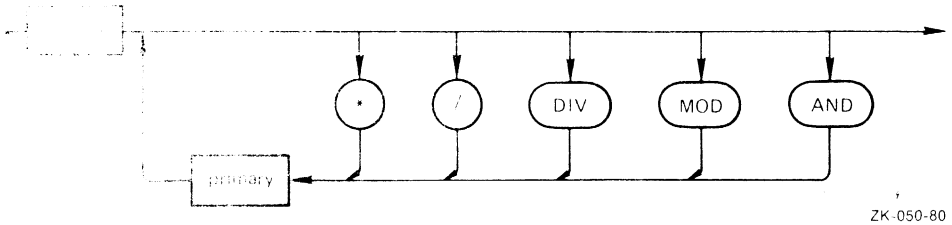


ZK-038-80

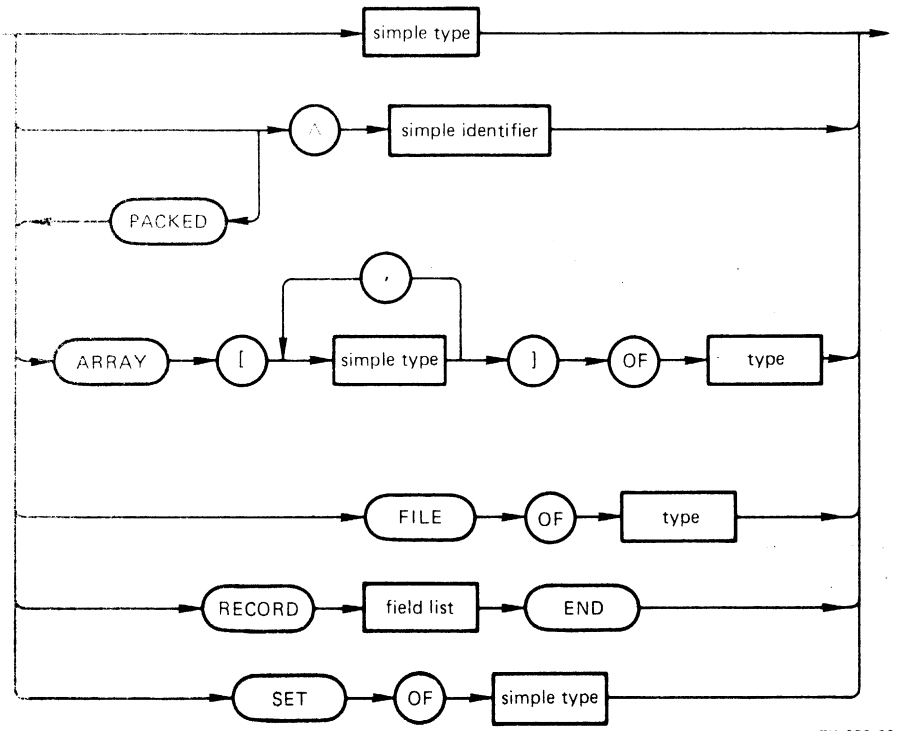
structured statement



ZK-043-80

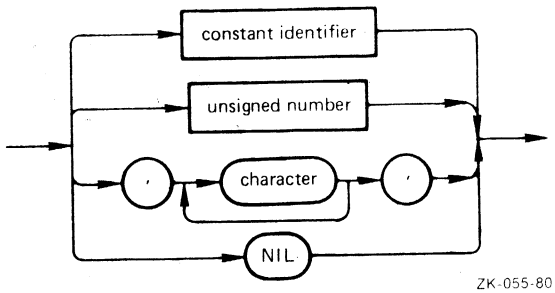


ZK-050-80



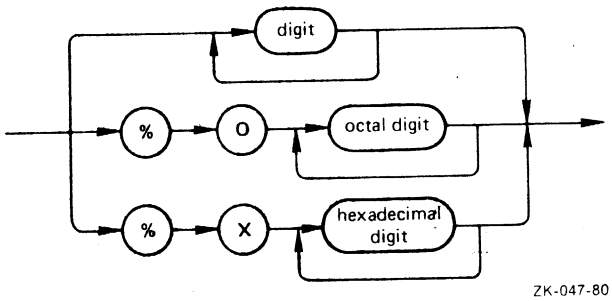
ZK-058-80

unsigned constant



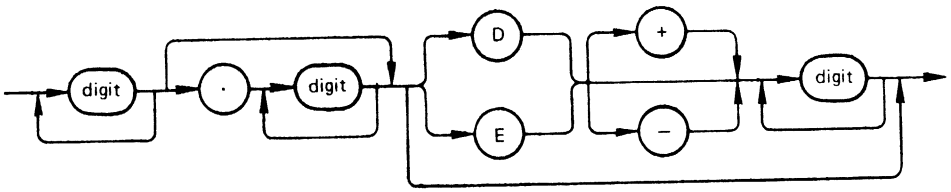
ZK-055-80

unsigned integer



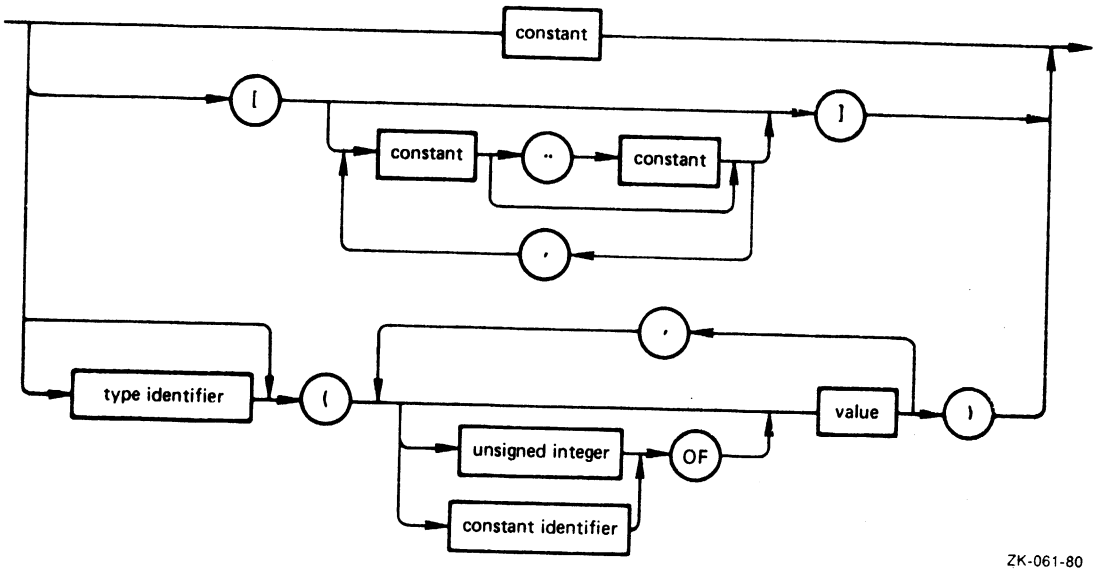
ZK-047-80

unsigned number



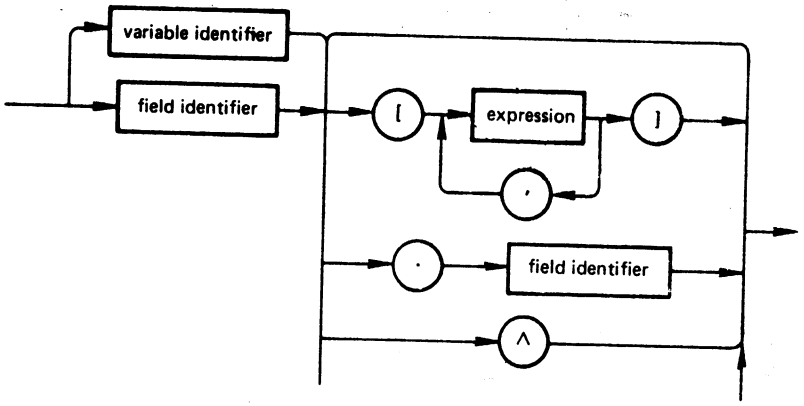
ZK-046-80

value



ZK-061-80

variable



ZK-060-80

APPENDIX C

SUMMARY OF VAX-11 PASCAL EXTENSIONS

Table C-1 summarizes the language features provided in VAX-11 PASCAL that are not defined in the PASCAL User Manual and Report.[1]

Table C-1
Language Extensions

Category	Extension
Lexical extensions	Identical treatment of lowercase and uppercase characters except in character and string constants and variables
	New reserved words: MODULE, OTHERWISE, VALUE, %DESCR, %STDESCR, %IMMED, %INCLUDE
	Exponentiation operator (**)
	Hexadecimal and octal constants
	Double-precision constants
	Dollar sign ($) and underscore (_) characters in identifiers
Predefined types	DOUBLE and SINGLE
Predefined procedures	CLOSE(f), DATE(a), FIND (f,n), HALT, LINELIMIT(f,n), OPEN(f,...), TIME(a)
Predefined functions	CARD(s), CLOCK, EXPO(r), LOWER(a,n), SNGL(d), UNDEFINED(r), UPPER (a,n)

(continued on next page)

1. K. Jensen and N. Wirth, PASCAL User Manual and Report, 2nd ed. (New York: Springer-Verlag, 1974).

Table C-1 (Cont.)
Language Extensions

Category	Extension
READ, READLN, WRITE, WRITELN extensions	Parameters of string and enumerated types for READ and READLN
	Parameters of enumerated types for WRITE and WRITELN
READ, READLN, WRITE, WRITELN extensions (Cont.)	Hexadecimal and octal output for WRITE and WRITELN
	Prompting at terminal with a WRITE/READ or WRITE/READLN sequence
	Optional carriage control specification for text files with WRITE and WRITELN
Program-level declarations	VALUE initialization in declaration section at program level
Statements	OTHERWISE clause in CASE statement
Procedures and functions	External procedure and function declarations
	Dynamic array parameters
	Extended passing mechanism specifiers for external procedures and functions: %IMMED, %DESCR, %STDESCR, %IMMED FUNCTION, %IMMED PROCEDURE
Compilation	MODULE capability for combining procedures and functions to be compiled separately from main program

APPENDIX D

SPECIFYING COMPILE-TIME QUALIFIERS IN THE SOURCE CODE

You can specify some compile-time qualifiers in the source code as listed in Table D-1. (For a complete description of compile-time qualifiers, see the VAX-11 PASCAL User's Guide.)

Table D-1
Compile-Time Qualifiers

Qualifier	Abbreviation	Default	Purpose
CHECK	C	C-	Generates code to perform run-time checks
CROSS_REFERENCE	X	X-	Produces a cross-reference listing of identifiers
DEBUG	D	D-	Generates some records for VAX-11 Symbolic Debugger
LIST	L	L-(interactive); L+(batch)	Produces source listing file
MACHINE_CODE	M	M-	Includes machine code listing in source listing file
STANDARD	S	S+	Prints messages indicating use of VAX-11 PASCAL extensions
WARNINGS	W	W+	Prints diagnostics for warning-level errors

SPECIFYING COMPILE-TIME QUALIFIERS IN THE SOURCE CODE

When specified in the source code, these qualifiers have the form:

```
(*$qualifier ,qualifier...  ;  comment*)
```

qualifier

Specifies a qualifier name or 1-character abbreviation.

comment

Denotes the text of a comment. The comment text is optional.

The first character after the comment delimiter must be a dollar sign ($); the dollar sign cannot be preceded by a space.

To enable a qualifier, place a plus sign (+) after its name or abbreviation. To disable a qualifier, place a minus sign (-) after its name or abbreviation. You can specify any number of qualifiers in a single comment. You can also include text in the comment after the qualifiers. The text must be separated from the list of qualifiers by a semicolon.

You can use qualifiers in the source code to enable and disable options during compilation. For example, to generate check code for only one procedure in a program, insert a comment that enables the CHECK qualifier before the procedure declaration. After the end of the procedure declaration, include a comment that disables the qualifier. For example:

```
(*$C+ ; enable CHECK for TEST1 only*)
PROCEDURE TEST1;
  .
  .
  .
END
(*$C-;disable CHECK option*)
```

You can specify qualifiers in both the source code and the PASCAL command line. Command line qualifiers override source code qualifiers. If, for example, the source code specifies DEBUG+, but you type PASCAL/NODEBUG, the DEBUG option is not in effect.

APPENDIX E

PROGRAM EXAMPLES

This appendix contains four programs that perform the following tasks:

Program 1 - Creates and loads a table

Program 2 - Counts the number of lines and characters in a file

Program 3 - Evaluates polynomials using Horner's rules

Program 4 - Alphabetizes a list of words entered on a terminal

TABLE SAMPLE

```
PROGRAM TABLE(INPUT, OUTPUT);
        CONST  Tablesize = 50;
        Blank = ' ';
        TYPE   Index = 0..Tablesize;
        VAR    R, T : Index;
        Truth : BOOLEAN;
        C, D  : CHAR;
        Inx   : INDEX;
        Table : PACKED ARRAY [1..Tablesize] OF CHAR;

PROCEDURE INIT_TABLE;
   BEGIN
     Inx := 0;
   END;

PROCEDURE PUTINTABLE
   (Ch : CHAR; VAR Pt : INDEX);
BEGIN
  IF Inx = Tablesize
  THEN
    BEGIN
      WRITELN ('TABLE OVERFLOW');
      HALT
      END
   ELSE
     BEGIN
       Inx := Inx + 1;
       Table[Inx] := Ch;
       Pt := Inx
     END;
END;

(*This function gets a character from the table*)

FUNCTION GetTable
         (pt : Index) : CHAR;
BEGIN
  GetTable := Table[Pt]
END;

PROCEDURE IsInTable
          (Ch : CHAR; VAR Yesno : BOOLEAN; VAR Pt : Index);
LABEL 1;
          VAR I : Index;
BEGIN
  Yesno := FALSE; Pt := 0;
  FOR I := 1 to Inx
  DO
    If Table[I] = Ch
    THEN
      BEGIN
        Yesno := TRUE;
        Pt := I;
        GOTO 1
      END;
1:
END;
```

```
BEGIN (*MAIN PROGRAM*)
  READ (C); Truth := FALSE; INIT_TABLE;

  REPEAT
    READ (C);
    IsInTable (C, Truth, T);
    IF Truth
    THEN
      WRITELN (C, 'IS ALREADY IN TABLE AT POSITION ', T:1)
    ELSE
    BEGIN
      PutInTable (C, T);
      WRITELN (blank, C, ' HAS BEEN ENTERED')
    END;
  UNTIL C = Blank;

  (*Prints contents of Table*)
  PAGE (OUTPUT);
  WRITELN ('TABLE POSITION' :16, 'TABLE CONTENTS' :16);
  WRITELN;
  FOR R := 1 TO Inx
    DO
     WRITELN (R:16, GetTable(R):16);
END.
```

E.2 TEXTCHECK

```
PROGRAM TEXT(INPUT, OUTPUT);
        CONST Max = 10000;
        VAR Lines, Letters, Digits, Spaces, Others, Characters, total: INTEGER;
        C:  CHAR;

BEGIN
  WRITE('INPUT NEW;'); WRITELN;
  Lines := 0; Letters := 0; Digits := 0; Spaces := 0; Others := 0;
  Characters := 0;
  WHILE NOT EOF
  DO
  BEGIN
    WRITE (' ');
    WHILE NOT EOLN(INPUT)
    DO
    BEGIN
      READ (C);
      WRITE(C);
      Characters := Characters + 1;
      IF Characters > Max
        THEN
      BEGIN
        WRITELN;
        WRITELN ('INPUT EXCEEDS', Max, 'CHARACTERS');
        HALT;
        END;
        IF ( C >= 'A') AND (C <= 'Z') OR (C >= 'a') AND
          (C <= 'z')
        THEN
           Letters := Letters + 1
        ELSE
           IF ( C >= '0') AND (C <= '9')
        THEN
           Digits := Digits + 1
        ELSE
           IF C = ' '
        THEN
           Spaces := Spaces + 1
        ELSE
           Others := Others + 1;
      END;
    WRITELN;
    READLN;
    Lines := Lines + 1;
  END;
  Total := spaces + others + letters + digits;
  IF Total <> Characters
   THEN
    WRITELN('0CHARACTER COUNT, ', Characters, ' is not equal to,',
     'the separate sum,', Total);
  WRITELN;
  WRITELN(' NUMBER OF INPUT LINES:  ', Lines);
  WRITELN(' NUMBER OF LETTERS: ', Letters);
  WRITELN(' NUMBER OF SPACES: ', Spaces);
  WRITELN(' NUMBER OF DIGITS: ', Digits);
  WRITELN(' NUMBER OF OTHER CHARACTERS: ', Others);
  WRITELN(' TOTAL NUMBER OF CHARACTERS: ', Characters)
END.
```

E.3 POLYNOMIALS

```
PROGRAM POLLY(INPUT,OUTPUT);
        VAR X, Degree, I : INTEGER;
        Coefficient:  ARRAY[0..20] OF INTEGER;
FUNCTION HORNER(X: INTEGER): INTEGER;
        VAR Temp, I: INTEGER;
BEGIN
  Temp := 0;
  FOR I := Degree DOWNTO 0 DO
       Temp := Temp * X + Coefficient[I];
  Horner := Temp;
END;

PROCEDURE Print_polly;
    VAR I: INTEGER;

BEGIN
  WRITELN;
  WRITE ('Polynomial:    ');

  FOR I := Degree DOWNTO 0 DO
     BEGIN
     WRITE (Coefficient[I]:1);
     IF (I <> 0)
     THEN
        WRITE (' X');
     IF (I > 1)
     THEN
        WRITE ('**', I:1);
     IF (I <> 0)
     THEN
        WRITE (' + ');
     END;

  WRITELN;
  WRITELN;
END;

BEGIN
  Degree := 0;
  WRITELN;
  WRITELN ('Begin typing coefficients.  Type a blank line when
done.');
  WRITE ('Enter coefficient of X**', Degree:1, ': ');

  WHILE NOT EOLN DO
     BEGIN
      READLN ( Coefficient[Degree]);
      Degree := Degree + 1;
      WRITE ('Enter coefficient of X**', Degree:1, ': ');
     END;
  READLN;        { This reads the empty line.    }

  Degree := Degree - 1;
```

```
  PRINT_POLLY;

  WRITE ('Begin typing values for the variable.');
  WRITELN ('  Type CTRL/Z when done.');
  WRITELN;
  WRITE ('Enter X Value : ');

  WHILE NOT EOF DO
     BEGIN
      READLN (X);
      WRITELN ('Polynomial = ', Horner(X):1, ' when X = ', X:1);
      WRITELN;
      WRITE ('Enter X Value : ');
     END;
END.
```

E.4 COUNTWORDS

```
PROGRAM COUNTWORDS(INPUT,OUTPUT,F);

CONST Word_Length = 20;

TYPE String        = PACKED ARRAY [1..Word_Length] OF CHAR;
     Ref_Tree_Node = ^Tree_Node;
(*                                    *
 *   Defines the tree                 *
 *                                    *)
     Tree_Node     = RECORD
                        Lower_Branch, Upper_Branch : Ref_Tree_Node;
                        Count                      : INTEGER;
                        Word                       : String
                     END;

VAR Root     : Ref_Tree_Node;
    New_Word : String;
    F        : TEXT;


FUNCTION Create_Node : Ref_Tree_Node;
    VAR New_Node : Ref_Tree_Node;

(*                                          *
 * Allocates storage and assigns an address  *
 *                                          *)
    BEGIN
    NEW(New_Node);
(*                                          *
 * Initializes the variables                *
 *                                          *)
    WITH New_Node^ DO
        BEGIN
        Lower_Branch:= NIL; Upper_Branch:= NIL; Count := 1; Word := New_Word;
        END;

    Create_Node := New_Node
    END;


(*                                         *
 *This procedure searches the tree until the*
 *word is located or until the new word    *
 *is inserted into the tree                *
 *                                         *)
PROCEDURE Enter_Node;
    VAR Current : Ref_Tree_Node;

    BEGIN
    Current := Root;

(*                                         *
 *Initializes the pointer Create_Node to the*
 *root of the tree                         *
 *                                          *)
```

```
    IF Current = NIL THEN
        Root := Create_Node
    ELSE
        REPEAT
            WITH Current^ DO
                IF Word = New_Word
                  THEN
(*                                      *
 *If the new word exists in the tree    *
 *the variable Count is incremented     *
 *by 1 and the pointer Current is set   *
 *to NIL                                *
 *                                      *)
                    BEGIN
                      Current := NIL;
                      Count := Count + 1
                    END
                ELSE
(*                                      *
 *The lower branch of the tree is       *
 *searched                              *
 *                                      *)
                     IF Word > New_Word
                  THEN
                    BEGIN
                      Current := Lower_Branch;
                    IF Current = NIL
                     THEN
                        Lower_Branch := Create_Node
                    END
                ELSE
(*                                      *
 * The upper branch of the tree is      *
 * searched                             *
 *                                      *)
                    BEGIN
                      Current := Upper_Branch;
                    IF Current = NIL THEN
                        Upper_Branch := Create_Node
                    END
        UNTIL Current = NIL;
    END;

(*                                      *
 *This procedure prompts for the input*
 *file name, opens the file and        *
 *performs a RESET                     *
 *                                     *)
PROCEDURE Initialize;
    VAR filename : PACKED ARRAY [1..32] OF CHAR;
    BEGIN
      WRITE('Enter the name of the file to be scanned: ');
      READLN(Filename);
      OPEN(FILE_VARIABLE := F, FILE_NAME := filename, HISTORY := OLD);
      RESET(F);
    END;
```

```
(*                                                *
 *This procedure may call itself to print the *
 *tree in alphabetical order                    *
 *                                                *)
PROCEDURE Print_Node (Current : Ref_Tree_Node);
    BEGIN
      IF Current <> NIL THEN
        WITH Current^ DO
            BEGIN
              Print_Node(Lower_Branch);
              WRITELN(Word,' ',Count:6);
              Print_Node(Upper_Branch)
            END
    END;


(*                                              *
 *This procedure scans the file for non-       *
 *alphabetics, makes lower case letters out *
 *of upper case letters and enters the word *
 *into the tree by calling Enter-Node          *
 *                                              *)

PROCEDURE Scan_File;
    VAR I : INTEGER;
        C : CHAR;
    BEGIN
    I := 0; C := ' ';

(*                                              *
 *checks for non alphabetics                   *
 *                                              *)
    WHILE NOT EOF(F) DO
        BEGIN
          WHILE NOT EOF(F) AND NOT (C IN ['A'..'Z','a'..'z']) DO
            READ(F,C);

        WHILE NOT EOF(F) AND (C IN ['A'..'Z','a'..'z']) DO
(*                                              *
 *Lower cases all upper case letters           *
 *                                              *)
            BEGIN
              IF C IN ['a'..'z'] THEN C := CHR(ORD(C) + ORD('A') - ORD('a'));
              I := I + 1;
              IF I <= Word_Length THEN New_Word[I] := C;
              READ(F,C)
            END;
```

```
(*                                              *
 *Enters the word into the tree via             *
 *the procedure                                 *
 *                                              *)
        IF I > 0 THEN
            BEGIN
              FOR I := I + 1 TO Word_Length DO
                New_Word[I] := ' ';
              Enter_Node;
              I := 0
            END
        END
    END;

(*                                              *
 *The main body of the program                  *
 *                                              *)

BEGIN
  Initialize;
  Scan_File;
  Print_Node(Root);
END.
```

APPENDIX F

VERSION 1.0 OPEN PROCEDURE

The following is the VAX-11 PASCAL Version 1.0 OPEN procedure. The syntax for this procedure has been changed for Version 1.2. However, Version 1.0 is still supported to allow users to run existing programs.

However, you should use the new syntax, as described in Section 7.1.5 whenever possible.

Section F.1 describes the Version 1.0 OPEN procedure.

Section F.2 describes the differences between Version 1.0 and 1.2.

F.1 THE OPEN PROCEDURE

The OPEN procedure opens a file and allows you to specify file attributes.

Format

```
OPEN( file-variable,
      [['VAX/VMS file-spec']] [[, buffer-size]] [[, file-status]] [[, record-access-mode]]
      [[, record-type]], carriage-control]]);
```

file-variable

Specifies the PASCAL file variable associated with the file being opened. You cannot open the predeclared file variable INPUT.

'VAX/VMS file-spec'

Provides information about the file for the operating system. You can use a VAX/VMS file specification or a logical name (see the VAX-11 PASCAL User's Guide for more information). The apostrophes are required to delimit the file specification or logical name. If you omit the file specification, PASCAL uses the default values shown in Table F-1.

The file variable and VAX/VMS file specification parameters designate the file to be opened. The remaining parameters specify attributes for the file. Except for the file variable name, all parameters are optional. Any parameters you specify, however, must be in the order shown above.

The OPEN procedure opens a file for access by the program. If the file does not exist, one is created with the specified name and attributes. You cannot use OPEN on a file that is already open, or on the predeclared file INPUT.

Table F-1
Default Values for VAX/VMS File Specifications

Element	Default
Node	Local computer
Device	Current user device
Directory	Current user directory
File name	PASCAL file variable name, truncated to first nine characters
File type	DAT
Version number	OLD: highest current number NEW: highest current number + 1

Because the RESET and REWRITE procedures implicitly open files, you need not always use the OPEN procedure. RESET and REWRITE impose the defaults for VAX/VMS file specification, buffer size, record access mode, record type, and carriage control shown in Tables F-1 and F-2. For the file status attribute, RESET uses a default of OLD, and REWRITE uses a default of NEW. You must use the OPEN procedure for the following:

- To create a file with fixed-length records
- To open a file for DIRECT access
- To specify a buffer size other than 133 for a text file

Table F-2
Summary of File Attributes

Parameter	Possible Values	Default
Buffer size	Any unsigned integer value	133 bytes
File status	OLD or NEW	NEW
Record access mode	DIRECT or SEQUENTIAL	SEQUENTIAL
Record type	FIXED or VARIABLE	VARIABLE for new file; for old file, record type established at file creation
Carriage control	LIST, CARRIAGE or NOCARRIAGE	LIST for all text files; NOCARRIAGE for all other files

F.1.1 Buffer Size

This unsigned integer specifies the maximum record size in bytes for a text file. The default value is 133 bytes. For a text file with variable length records, the buffer size value denotes the maximum number of characters on a line. The buffer size parameter applies only to text files. For any other type of file, the buffer size has no meaning and is ignored.

By default, text files have variable-length records. If you create a text file with fixed-length records, the buffer size specifies the exact length of the records. You must read the file with the buffer size you specified at its creation.

F.1.2 File Status -- NEW or OLD

The file status indicates whether the specified file exists or must be created. A file status of NEW indicates that a new file must be created with the specified attributes. NEW is the default value.

If you specify OLD, the system tries to open an existing file. An error occurs if the file cannot be found. OLD is invalid for internal files, which are newly created each time the declaring program or subprogram is executed.

F.1.3 Record Access Mode -- SEQUENTIAL or DIRECT

The record access mode specifies sequential or direct access to the components of the file. In SEQUENTIAL mode, you can access files with fixed- or variable-length records. The default access mode is SEQUENTIAL.

DIRECT mode allows you to use the FIND procedure to access files with fixed-length records. You cannot access a file with variable-length records in DIRECT mode.

F.1.4 Record Type -- FIXED or VARIABLE

The record type specifies the structure of the records in a file. A value of FIXED indicates that all records in the file have the same length. A value of VARIABLE indicates that the records in the file can vary in length. VARIABLE is the default for a new file. For an existing file, the default is the record type associated with the file at its creation.

F.1.5 Carriage Control -- LIST, CARRIAGE, or NOCARRIAGE

The carriage control option specifies the carriage control format for a text file. A value of LIST indicates single spacing between components. LIST is the default for all text files, including the predefined file OUTPUT.

The CARRIAGE option indicates that the first character of every output line is a carriage control character.

NOCARRIAGE means that no carriage control applies to the file.

You cannot create a file with one carriage control format and later write it with another.

F.1.6 Examples

1.
```
VAR Userguide : TEXT;
  .
  .
  .
OPEN (Userguide);
```

   When the OPEN procedure is executed, the system first attempts to use Userguide as a logical name. If no such logical name is assigned, it creates the file Userguide.DAT in your default device and directory on the local computer. By default, the file is created with a buffer size of 133 bytes and records of variable length. The system then opens the file for sequential access.

2.
```
OPEN (Userguide, 80);
```

   This statement causes the system to create Userguide.DAT as in the previous example, except that the buffer size is 80 bytes instead of 133 bytes.

3.
```
OPEN (OUTPUT, CARRIAGE);
```

   This example causes the system to interpret the first character of each line written to the predeclared file OUTPUT as a carriage control character. When you call OPEN for the predeclared file OUTPUT, you can specify only a carriage control option. If you include any other parameters, an error occurs.

4.
```
OPEN (Albums, 'DB1:[EASTWEST]INVENT', OLD, DIRECT);
```

   This example opens the existing VAX/VMS file DB1:[EASTWEST]INVENT.DAT for direct access. The VAX/VMS file is known to the PASCAL program as the file variable Albums.

5.
```
OPEN (Solar, 'Energy', NEW, FIXED);
```

   Assuming that ENERGY is defined as a logical name, this statement creates a file with the VAX/VMS specification designated by the logical name ENERGY. The file is created with fixed-length records.

F.2 DIFFERENCES IN OPEN SYNTAX

The following differences exist between the Version 1.0 and 1.2 OPEN procedure syntax.

- In Version 1.0 the file name was specified as a literal string constant (VAX/VMS file specification) or a logical name. In Version 1.2 a variable containing a file specification can be used as the file name.

VERSION 1.0 OPEN PROCEDURE

- In Version 1.2 the following parameters have been renamed:

Version 1.0	Version 1.2
file status	history
buffer-size	record-length

- In Version 1.2 the parameters buffer-size (now record-length) and file -status (now history) have been reversed.
- In Version 1.2 CARRIAGE_CONTROL has two new values:
  - FORTRAN, which is equivalent to CARRIAGE
  - NONE, which is equivalent to NOCARRIAGE
- In Version 1.2 all parameters are nonpositional when specified by using a parameter name.

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z

VAX-11 PASCAL
Language Reference
Manual
AA-H484B-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 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

Please cut along this line.

— — 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/J3-5
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

— — Do Not Tear - Fold Here —