VAX-11 PASCAL
Language Reference Manual

Order No. AA-H484C-TE

October 1982

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.

SUPERSESSION/UPDATE INFORMATION: This revised document supersedes the VAX-11 PASCAL Language Reference Manual (Order No. AA-H484B-TE)

SOFTWARE VERSION: VAX-11 PASCAL V2.0

digital equipment corporation · maynard, massachusetts

First Printing, November 1979
Revised, March 1981
Revised, October 1982

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

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

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

Printed in U.S.A.

A postpaid READER'S COMMENTS form is included on the last page of this document. Your comments will assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

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

ZK2085

HOW TO ORDER ADDITIONAL DOCUMENTATION

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

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

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

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

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

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

DIRECT MAIL ORDERS (CANADA)

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

DIRECT MAIL ORDERS (INTERNATIONAL)

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

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

Contents

Chapter 6 Procedures and Functions

Chapter 7 Predeclared Routines

Chapter 8 Input and Output

Chapter 9 Compilation Units

Chapter 10 Attributes

Figures

Tables

Preface

Manual Objectives

This manual describes the VAX-11 PASCAL language, which is an extension of the standard proposed for the PASCAL programming language by the International Organization for Standardization. This manual is designed primarily for reference; it is not a tutorial document. For information about tutorial and user documents, refer to the Associated Documents list later in this preface.

Intended Audience

Readers who know the PASCAL language will benefit most from this manual. You need not have a detailed understanding of the VAX/VMS operating system, but some familiarity with VAX/VMS is helpful. Relevant documents about VAX/VMS are also listed under Associated Documents.

Structure Of This Document

This manual has 10 chapters and 8 appendixes.

- Chapter 1 contains an overview of the VAX-11 PASCAL language and illustrates the structure of a PASCAL program.
- Chapter 2 provides detailed information on data types.
- Chapter 3 discusses expressions involving constants, variables, function designators, and operators.
- Chapter 4 describes the declaration sections.
- Chapter 5 explains the statements that perform the actions of a program.
- Chapter 6 discusses how to write procedures and functions.
- Chapter 7 presents the predeclared procedures and functions supplied by VAX-11 PASCAL.
- Chapter 8 provides detailed information on input and output procedures.
- Chapter 9 describes compilation units and independent compilation.
- Chapter 10 provides information on VAX-11 PASCAL attributes.
- Appendix A lists the ASCII character set.
- Appendix B presents the syntax productions and diagrams for the VAX-11 PASCAL language.
- Appendix C summarizes the predeclared procedures and functions available in VAX-11 PASCAL.
- Appendix D lists the extensions incorporated in VAX-11 PASCAL.
- Appendix E explains the differences between Version 2 and previous versions of VAX-11 PASCAL.
- Appendix F describes how the VAX-11 PASCAL compiler and run-time system detect violations of the language standard.
- Appendix G describes the features of PASCAL that are defined by or dependent on the VAX-11 implementation.
- Appendix H gives complete PASCAL program examples.

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 that will help all users in creating, editing, copying, and printing files containing PASCAL programs.

The *VAX-11 Information Directory and Index* briefly describes all VAX/VMS 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: {expression} {statement} ... A horizontal ellipsis means that the item preceding the ellipsis can be repeated; for example: digit ...
{ },...	Braces followed by a comma and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with commas; for example: {label},...
{ };...	Braces followed by a semicolon and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with semicolons; for example: REPEAT {statement};... UNTIL expression
. . .	A vertical ellipsis in a figure or example means that not all of the statements are shown.

[ ]	Square brackets mean that the statement syntax requires the square bracket characters. This notation is used with arrays, sets, and attribute lists; for example: `ARRAY[index1]`
[[ ]]	Double brackets enclose items that are optional; for example: `EOLN [[ ]]`

items in UPPERCASE letters and special symbols	Uppercase letters and special symbols in syntax descriptions indicate VAX-11 PASCAL reserved words and predeclared identifiers; for example:
	
	`BEGIN`
	`END`

items in lowercase letters	Lowercase letters represent elements that you must replace according to the description in the text.

In this manual, complex examples and syntax diagrams have been divided into several lines to make them easy to read. PASCAL does not require that you format your programs in any particular way; therefore, you should not regard the formats used in this manual as mandatory.

Chapter 1
Introduction

VAX-11 PASCAL is an extended implementation of the PASCAL language that has been developed for use under the VAX/VMS operating system. It includes all the standard language elements plus the following extensions:

- UNSIGNED data type
- Double- and quadruple-precision real data types
- VARYING OF CHAR structured data type for items that can accept character strings of varying lengths
- Exponentiation operator
- Initialization of variables in a VAR section
- OTHERWISE clause in the CASE statement
- Extended parameter specifications
- Extended input and output capabilities, including support for relative and indexed file organizations
- Independent compilation
- Attributes that modify data items and the names of procedures, functions, programs, and modules

In this manual, the term "VAX-11 PASCAL" is used to emphasize features that are found in the VAX-11 implementation but not in the PASCAL language definition.

This chapter presents an overview of PASCAL, including some VAX-11 extensions, and illustrates the structure of a PASCAL program. It also describes PASCAL's lexical elements—the character set, reserved words, identifiers, and special symbols. The final sections explain how to document a program and how to include existing files in a source program.

1.1 Overview of VAX-11 PASCAL

A PASCAL program performs operations on data items known as constants, variables, and function results. A constant is a quantity with an unchanging value; a constant to which you give a name is called a symbolic constant. A variable is a quantity whose value can change while the program executes. A function result is the value returned following the execution of a function.

1.1.1 Data Types

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

PASCAL provides identifiers for many predefined types. Thus, a program's operations can involve integers, real numbers, Boolean and character data, records, arrays, character strings, sets, files, and pointers to dynamic variables. VAX-11 PASCAL includes another predefined type, which you can use to represent large unsigned integers. PASCAL also allows you to create your own types by defining an identifier of your choice to represent a range of values; a user-defined type is also associated with a set of operators and a storage requirement.

The type of a constant is the type of its corresponding value. The type of a variable is the type established when the variable is declared and generally cannot be changed. The type of a function result is the type of the value returned by the function (called the result type).

Variables and function results can change in value any number of times. However, all of the values they assume must be within the range established by their type. A variable does not assume a value until the program assigns it one. A function result is computed during the execution of the function.

In PASCAL, types are associated not only with data items but also with expressions. An expression represents the computation of a value resulting from a combination of variables, constants, function results, and operators. You can use arithmetic, relational, logical, string, and set operators to form PASCAL expressions. Arithmetic operations produce integer, unsigned, or real-number values. Relational and logical operations yield Boolean results. String operations manipulate strings of characters. Set operations form the union, intersection, and differences of two sets.

1.1.2 Definitions and Declarations

PASCAL requires that you define every symbolic constant and user-created type and declare every label, variable, procedure, and function used in a program. You define and declare such data in the declaration section of the program, which can contain LABEL, CONST, TYPE, VAR, PROCEDURE, and FUNCTION sections. All of these sections except LABEL introduce identifiers and indicate what they represent; a LABEL section declares numeric

labels that correspond to executable statements accessed by GOTO statements. In VAX-11 PASCAL, a VAR section can assign initial values to the variables declared. An initialized variable assumes the given value when program execution begins.

1.1.3 Executable Statements

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

1.1.4 Routines

PASCAL allows you to group definitions, declarations, and executable statements into routines. You can use routines as a convenient way to organize a program by isolating the individual tasks that the program is to accomplish.

PASCAL has two kinds of routines—procedures and functions. Procedures are usually written to perform a series of actions. They are called by an executable statement known as a procedure call. Functions are written to compute and return a value; they are called when a function designator appears within an expression. PASCAL supplies many predeclared routines that perform frequently used operations, such as input and output.

Normally, a routine consists of a heading and a block, which you supply in the routine's declaration. The heading provides the routine's name, usually a list of formal parameters that declare the external data for the routine, and, in the case of functions, the type of the function result. The routine block consists of an optional declaration section and an executable section. The purpose of a declaration section in a routine is to declare data items that are local to the routine (that is, data items that are unavailable outside the routine).

1.1.5 Scope of Identifiers

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

The part of the program in which you have access to an identifier is called the scope of the identifier. Outside its scope, an identifier has either no meaning or a different meaning. Specifically, the scope of an identifier is the block in which it is declared. Since blocks can be nested, the scope of a particular identifier can include blocks at lower levels in the program hierarchy. You must keep track of the scope of identifiers, especially if you plan to use the same name for several data items.

1.1.6 Compilation Units

VAX-11 PASCAL uses the term "compilation unit" to denote either a program or a module, each of which can be compiled as a separate unit (unlike a routine, which cannot be compiled without the context of a program or module). A program consists of a heading and a block, just as a routine does. A VAX-11 PASCAL module consists of a heading followed only by a declaration section; it cannot contain executable statements. The heading contains the name of the program or module and, possibly, a list of identifiers that indicate any external files used. The data items declared in a compilation unit are available at all levels of the compilation unit, including nested routines, and are also available to subsequently compiled programs and modules, which can "inherit" these declarations.

1.1.7 Attributes

The VAX/VMS operating system controls how a VAX-11 PASCAL program is compiled, linked, and executed. The defaults provided by the various components of VAX/VMS are sufficient for most applications; however, for advanced applications, such as systems programming, you may need to change such factors as the allocation size, addressing boundaries, and form of storage occupied by variables; the techniques used by the VAX-11 PASCAL compiler to compile your program; and the sharing of data declarations among compilation units. By including a class of language extensions known as attributes, VAX-11 PASCAL allows you to change many of the properties of a program that are normally determined by VAX/VMS.

Attributes are identifiers that specify how variables, formal parameters, routines, and compilation units are to be qualified by the changes you make to VAX/VMS defaults. The syntax for specifying attributes is given throughout this manual in the sections describing type definitions, variable declarations, and routine, program, and module headings. Explanations, rules, and defaults for all the attributes are provided in Chapter 10.

1.1.8 Structure of a PASCAL Program

Figure 1-1 illustrates some of the typical parts of a PASCAL program.

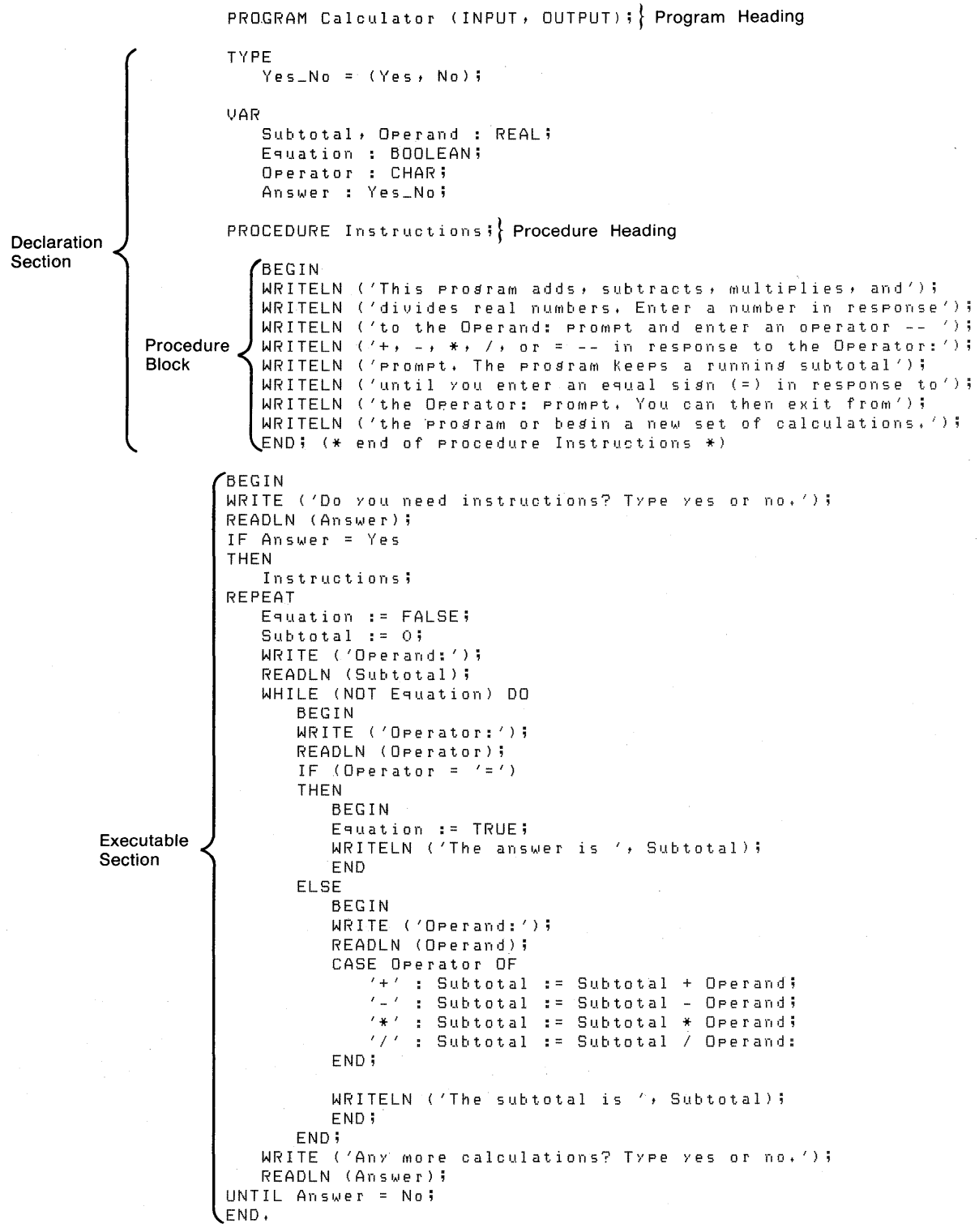


Figure 1–1: Structure of a PASCAL Program

1.2 Lexical Elements

A PASCAL program is composed entirely of lexical elements. These elements may be individual symbols, such as arithmetic operators, or they may be words that have special meanings in PASCAL. The basic unit of any lexical element is a character, which must be a member of the ASCII character set, as described in Section 1.2.1. Some characters are special symbols that are used in PASCAL as statement delimiters, operators, and elements of the language syntax. The special symbols used in VAX-11 PASCAL are presented in Section 1.2.2.

The words used in a PASCAL program are combinations of alphabetic and numeric characters and occasionally a dollar sign ($), an underscore (_), or a percent sign (%). Some words are reserved for the names of executable statements, operations, and predefined data structures. The words that are reserved in VAX-11 PASCAL are listed in Section 1.2.3. Other words in a PASCAL program are identifiers. Predeclared identifiers represent routines and data types provided by VAX-11 PASCAL. Other identifiers are created by the user to name programs, symbolic constants, variables, and any necessary program elements that have not already been named. Section 1.2.4 explains how to use both kinds of identifiers in a program.

1.2.1 Character Set

VAX-11 PASCAL uses an extended American Standard Code for Information Interchange (ASCII) character set (see Appendix A). This extended ASCII character set contains 256 characters, each of which corresponds to a numeric value. The characters fall into the following categories:

- The upper- and lowercase letters A through Z and a through z
- The numbers 0 through 9
- Special characters, such as the ampersand (&), question mark (?), and equal sign (=)
- Nonprinting characters, such as the space, tab, line feed, carriage return, and bell
- Extended, unspecified characters with numeric codes from 128 to 255

The VAX-11 PASCAL compiler does not distinguish between upper- and lowercase characters except when they appear inside apostrophes. For example, the word PROGRAM has the same meaning when written as any of the following:

```
PROGRAM

PRoGrAm

program
```

The characters below, however, represent different values:

```
'b'

'B'
```

Similarly, the following two phrases represent different values:

```
'BREAD AND ROSES'

'Bread and Roses'
```

1.2.2 Special Symbols

Special symbols represent delimiters, operators, and other syntactic elements. VAX-11 PASCAL's special symbols are listed in Table 1-1. In symbols composed of more than one character, the characters cannot be separated by spaces.

Table 1-1: Special Symbols

Name	Symbol	Name	Symbol
Apostrophe	'	Less than	<
Assignment operator	:=	Less than or equal	<=
Brackets	[ ]	Minus sign	-
	(. .)	Multiplication	*
Colon	:	Not equal	<>
Comma	,	Parentheses	( )
Comments	(* *)	Percent	%
	{ }	Period	.
Division	/	Plus sign	+
Equal	=	Pointer	^
Exponentiation	**		@
Greater than	>	Semicolon	;
Greater than or equal	>=	Subrange operator	..
		Type cast operator	::

1.2.3 Reserved Words

In the PASCAL language definition, the words in Table 1-2 are reserved for the names of statements, data types, and operators. This manual shows these words in uppercase letters.

Table 1-2: Standard Reserved Words

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

You can use reserved words in your program only in the contexts for which they are defined. You cannot redefine a reserved word for use as an identifier.

The nonstandard words listed in Table 1–3 are reserved for VAX–11 PASCAL extensions. If you wish, you may redeclare those words that do not contain a percent sign (%); however, any extension using those words becomes unavailable within the block in which the word was redeclared. Nonstandard words beginning with the percent sign may not be redeclared as identifiers because they contain a special symbol. This manual shows nonstandard reserved words in uppercase letters.

Table 1–3: Nonstandard Reserved Words

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

1.2.4 Identifiers

In PASCAL, identifiers are used to name programs, modules, symbolic constants, data types, variables, procedures, functions, and program sections. An identifier is a combination of letters, digits, dollar signs ($), and underscores (_); it must conform to the following restrictions:

- An identifier cannot start with a digit.
- An identifier cannot contain any spaces or special symbols.
- The first 31 characters of an identifier must denote a unique name within the block in which the identifier is declared.

In VAX–11 PASCAL, only the first 31 characters of an identifier are scanned for uniqueness. A warning message results from every occurrence of an identifier that exceeds 31 characters. The following examples show 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)
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.

1.2.4.1 Predeclared Identifiers — VAX-11 PASCAL predeclares the identifiers listed in Table 1-4 as the names of procedures, functions, data types, symbolic constants, and file variables. Predeclared identifiers appear in uppercase letters throughout this manual.

Table 1-4: Predeclared Identifiers

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

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

For example, the predeclared identifier READ denotes the READ procedure, which performs input operations. If you use the word "read" to denote something else, perhaps a variable, you cannot use the READ procedure within the same block. You should avoid redefining predeclared identifiers because you could lose access to useful language features.

1.2.4.2 User Identifiers — User identifiers denote the names of programs, modules, symbolic constants, variables, procedures, functions, program sections, 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 Comments

In addition to data declarations and executable statements, a PASCAL program can contain comments—words and phrases that record important information about the program. When processing a program, the compiler ignores

the text of a comment; therefore, a comment can contain any ASCII character (except a nonprinting control character) and can appear anywhere a space is legal.

To signify a comment, you can either enclose the text in braces or precede it with a left-parenthesis/asterisk character pair and follow it with an asterisk/right-parenthesis character pair. For example:

```
{ This is a comment. }

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

In VAX-11 PASCAL, the special symbols used to delimit comments are equivalent. Thus, once you have begun a comment with an opening delimiter, the first occurrence of a closing delimiter of either kind ends the comment. For example:

```
{ The delimiters of this comment do not match. *)

(* PASCAL allows you to mix delimiters in this way. }
```

However, VAX-11 PASCAL does not allow you to nest comments. That is, you cannot include one set of comments within another. For example:

```
(* Comments cannot be nested { contained inside more than one
   set of comment delimiters } within your program. *)
```

The above example would result in a compile-time error.

1.4 The %INCLUDE Directive

The %INCLUDE directive allows you to access the text from one PASCAL source file during the compilation of another; the directive is useful when the same information is used by several programs. The contents of the included file are inserted at the point where the compiler encounters the %INCLUDE directive. This directive can appear anywhere that a comment is legal.

Syntax

%INCLUDE 'VAX/VMS file-specification [[{ /LIST | /NOLIST }]] ;

VAX/VMS file-specification

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

/LIST

An option that indicates that the included file should be printed in the listing of the program if a listing is being generated. This option is the default.

/NOLIST

An option that indicates that the included file should not be printed in the listing of the program. However, the line containing the %INCLUDE directive does appear in the program listing if one is being generated.

When the compiler finds the %INCLUDE directive, it stops reading from the current file and begins reading from the included file. When the compiler reaches the end of the included file, it resumes compilation at the point in the original file following the line that contains the %INCLUDE directive. If you specify neither /LIST nor /NOLIST, the source listing state (that is, whether or not a source listing is being produced) does not change when the compiler switches to the included file.

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, Architecture,
                            Agriculture, 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 though it were written as follows:

```
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, Architecture,
                          Agriculture, Hotel);
               END;
   .
   .
   .
```

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

A file included at the outermost level of a program is said to be included at the first level. A file included by a first-level file is said to be included at the second level, and so on. In general, a program may not include any files beyond the fifth level. Nesting levels may be further restricted by the number of open files that you as a user of your system are allowed to have open at one time. Figure 1–2 illustrates the legal levels of included files.

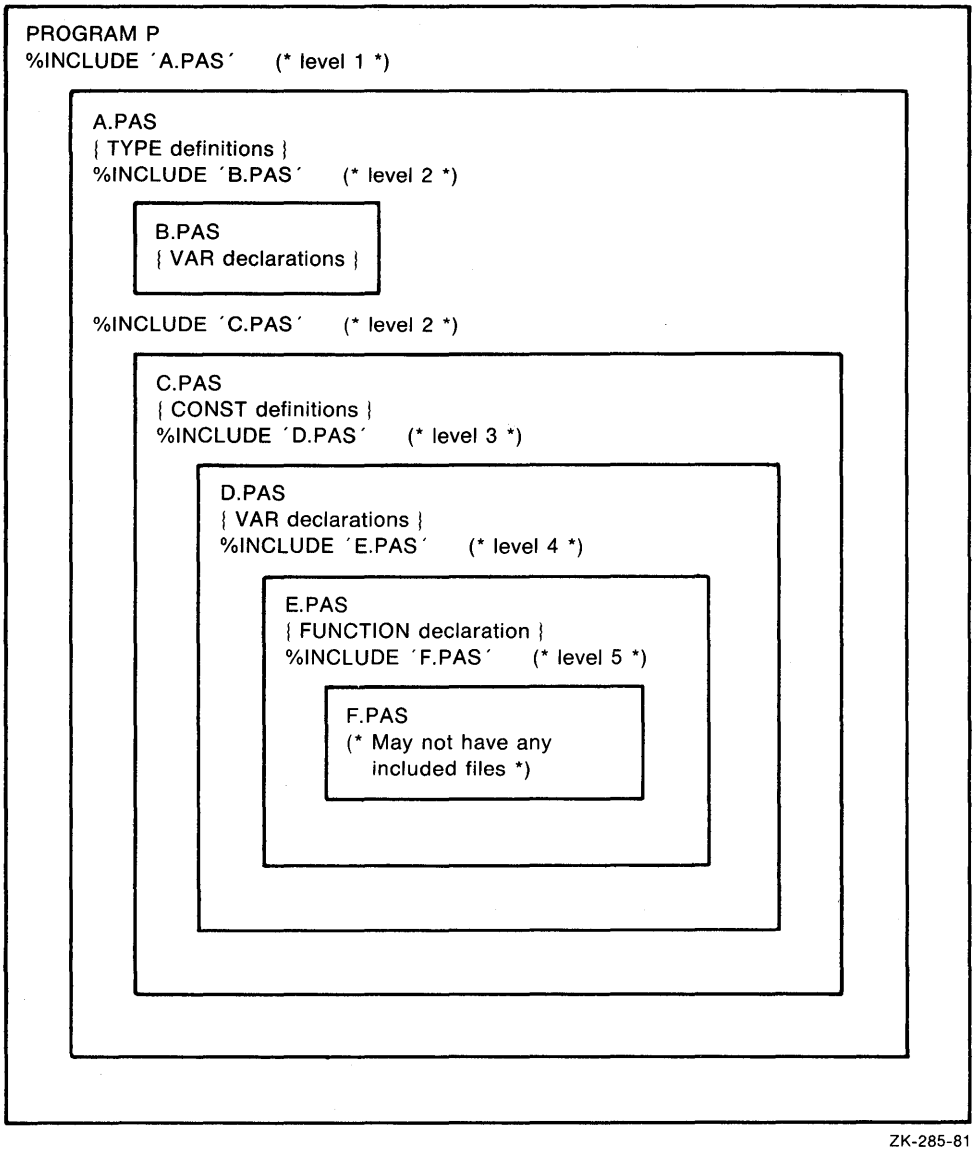


Figure 1–2: %INCLUDE File Levels

Chapter 2
Data Types

VAX-11 PASCAL has four categories of data types: ordinal, real, structured, and pointer. Ordinal and real types, which are often referred to collectively as the scalar types, are the fundamental types that serve as building blocks for the structured types. The pointer type allows you to refer to dynamically allocated variables.

VAX-11 PASCAL supplies predefined ordinal types for integer, character, and Boolean data. Two predefined types denote integer values. The type INTEGER represents signed integer values; the type UNSIGNED represents nonnegative values of the VAX-specific logical unsigned type (refer to the *VAX Architecture Handbook* for a full description of this type). The type CHAR signifies individual alphabetic, numeric, and special characters. The type BOOLEAN consists of the values FALSE and TRUE.

In addition, PASCAL allows you to define your own ordinal types in one of two ways:

- By enumerating each value of the type (called an enumerated type)
- By defining the type as a subrange of another ordinal type (called a subrange type)

Three predefined real types provide explicit single-, double-, and quadruple-precision real numbers.

VAX-11 PASCAL has five structured types: RECORD, ARRAY, VARYING OF CHAR, SET, and FILE. Structured types allow you to process groups of ordinal, real, structured, and pointer data items. For example, you could have a varying-length string of characters, a file of records, or an array of pointers.

The pointer type consists of the storage addresses of dynamic variables and the constant identifier NIL.

This chapter is organized as follows:

- Section 2.1 discusses the ordinal types—INTEGER, UNSIGNED, CHAR, BOOLEAN, enumerated, and subrange.
- Section 2.2 discusses the real types—REAL, SINGLE, DOUBLE, and QUADRUPLE.

- Section 2.3 discusses the structured types—RECORD, ARRAY, VARYING OF CHAR, SET, and FILE.
- Section 2.4 discusses the pointer type.
- Section 2.5 presents the rules of type compatibility, which determine the operations and assignments you can perform with data items of different types.

2.1 Ordinal Types

The values in an ordinal type have a one-to-one correspondence with the set of positive integers. These values are ordered so that each has a unique ordinal value that indicates its position in a list of all the values of the type. The ordinal types are discussed individually in Sections 2.1.1 through 2.1.6.

Three predeclared functions operate only on expressions of an ordinal type; they return information about the type's ordered sequence of values. The PRED function finds the predecessor of any value of an ordinal type (except the smallest). Similarly, the SUCC function finds the successor of any value of an ordinal type (except the largest). The ORD function finds the ordinal value of any value of an ordinal type and returns it as an integer. Note that the ordinal value of an integer is the integer itself. Chapter 7 provides further information on these functions.

2.1.1 INTEGER Type

The INTEGER data type denotes positive and negative integer values ranging from -2**31+1 through 2**31-1. This range contains numbers from -2,147,483,647 through 2,147,483,647. The largest possible value of the INTEGER type is known by the predefined constant identifier MAXINT.

You indicate a decimal integer by using decimal digits. No commas or decimal points are allowed. The following are valid decimal integers in PASCAL:

```
   17
    0
89324
```

VAX-11 PASCAL also allows you to specify integers in binary, octal, and hexadecimal notations. You can use integers written in these notations anywhere that decimal integers are permitted (except as labels; see Section 4.1). To specify an integer in binary, octal, or hexadecimal notation, place a percent sign (%) and a letter in front of a number enclosed in apostrophes. The appropriate letters, which may be either upper- or lowercase, are B for binary notation, O for octal notation, and X for hexadecimal notation. Inside the apostrophes, you can include spaces and tabs to make the notation easy to read. Note that regardless of which notation you use, the integer value may not be greater than MAXINT nor less than -MAXINT. For example:

```
%b'1000 0011'
%O'7712'
%x'DEC'
```

You can use negative integers in binary, octal, decimal, and hexadecimal notations. However, a negative integer such as -27 is not a constant, but is actually an expression consisting of the negation operator (-) and the integer value 27. The use of negative integers in complex expressions may not produce the results you expect; see Section 3.2.1 for more explanation. The input operations described in Chapter 8 allow you to supply a leading plus or minus sign with integer values; output operations, also described in Chapter 8, automatically supply leading minus signs with negative integer values.

2.1.2 UNSIGNED Type

The UNSIGNED data type denotes nonnegative integer values from 0 through 2**32-1. The largest possible value of the UNSIGNED data type is 4,294,967,295, which is more than twice as large as the value of MAXINT. UNSIGNED is a machine-dependent type intended for use in systems programming, not for every application involving nonnegative integers.

When a VAX-11 PASCAL program contains an integer constant greater than MAXINT or less than -MAXINT, the constant is treated as being of type UNSIGNED. Unsigned integers can be written in decimal, binary, octal, and hexadecimal notations (see Section 2.1.1 for notation rules). Integer constants not greater than MAXINT and not less than -MAXINT are always treated as being of type INTEGER.

2.1.3 CHAR Type

The CHAR data type comprises single character values from the ASCII character set, as listed in Appendix A. To specify a character constant, enclose a printable ASCII character in apostrophes. The apostrophe character itself must be typed twice within apostrophes. Each of the following is a valid character constant:

```
'A'
'z'
'0'
'+'
''''
'?'
```

You can write character strings such as 'HELLO' and '****', but you must represent them as packed arrays of characters (see Section 2.3.2.2) or varying-length character strings (see Section 2.3.3).

When you use the ORD function on an expression of type CHAR, the result is the ordinal value of the character in the ASCII character set. For example, if the variable Q_Char has the value 'Q', then the expression

```
ORD (Q_Char)
```

returns the integer 81, which is the ordinal value of uppercase Q in the ASCII character set.

The order of the characters in the ASCII character set may not be what you expect if you are not familiar with the set. 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 specify a nonprinting character such as a control character by writing an empty string, '', followed immediately by the ordinal value of the character in the ASCII character set, enclosed in parentheses. For example:

```
''(7)
```

This constant represents the control character that corresponds to the bell on your terminal.

2.1.4 BOOLEAN Type

The BOOLEAN data type consists of two constant values denoted by the predeclared identifiers FALSE and TRUE. These values are ordered so that FALSE is less than TRUE. Thus, the ORD function applied to the Boolean value FALSE returns the integer 0; ORD (TRUE) returns the integer 1.

Boolean values are the result of testing relationships for truth or validity.

2.1.5 Enumerated Type

An enumerated type is an ordered set of constant values denoted by identifiers. The enumerated type syntax requires that all constant identifiers of the type be listed in order and enclosed in parentheses.

Syntax

({identifier},...)

identifier

A constant value of the type.

The values of an enumerated type follow a left-to-right order such that any identifier in the list has an ordinal value greater than the ordinal values of all identifiers to its left and less than the ordinal values of all identifiers to its right. Thus, given:

```
(Spring, Summer, Fall, Winter)
```

Spring is less than Fall because Spring precedes Fall in the list of constant values.

The definition of an enumerated type associates an ordinal value with each identifier. The ordinal value of the first identifier is 0; the ordinal value of the second identifier is 1, and so forth. You can apply the ORD function to expressions of enumerated types. Using the example above, the expression ORD (Summer) is legal. Its result is 1 because Summer is the second value listed.

An identifier in an emumerated type cannot be defined for any other purpose. For example, the following enumerated type:

```
(Fall, Winter, Spring)
```

cannot be defined in the same block as the previous type because the identifiers Spring, Fall, and Winter would not be unique. Since the result of ORD (Fall) could be either 2 or 0, it is in fact undefined.

A maximum of 65,535 identifiers can be listed in an enumerated type.

Some examples of enumerated types are:

```
(Milk, Water, Cola, Beer)
(Swim, Run, Ski)
(Oatmeal, Sugar, Peanut_Butter, Choc_Chip)
```

2.1.6 Subrange Type

A subrange type specifies a limited portion of another ordinal type (called the base type) for use as a distinct type. The subrange syntax indicates the lower and upper limits of the type.

Syntax

lower-bound..upper-bound

lower-bound

A constant expression that establishes the lower limit of the subrange.

upper-bound

A constant expression that establishes the upper limit of the subrange.

The subrange type is defined only for the values between and including the lower and upper bounds. The value of the upper bound must be greater than or equal to the value of the lower bound. The subrange symbol (..) separates the bounds of the subrange.

The base type can be any enumerated or predefined ordinal type. The values in the subrange type are in the same order as they are in the base type. For example, the result of the ORD function applied to a value of a subrange type is the ordinal value that is associated with the relative position of the value in the base type, not in the subrange type.

You can use a subrange type anywhere in a program that its base type is legal. A value of a subrange type is converted to a value of its base type before it is used in an operation. All rules that govern the operations performed on an ordinal type pertain to subranges of the type.

The use of subrange types can make a program clearer. For example, you can limit the legal values for the days of the year by defining the subrange type 1..366.

If you enable subrange checking at compile time, the system generates a run-time error for the assignment of an out-of-range value to a subrange variable. In the above example, such an error occurs when an integer value less than 1 or greater than 366 is assigned to a variable of the subrange type. If you do not enable subrange checking, the compiler does not detect invalid assignments to

subrange variables. (See Section 10.5 and the *VAX-11 PASCAL User's Guide* for more information about subrange checking.)

The following are examples of subrange types and some possible uses for them:

```
'0'..'9'        (* single-digit numbers *)

'A'..'M'        (* the first half of the alphabet *)

1..31           (* the days of a month *)

Jan..Jun
May..Dec        (* given an enumerated type
                   listing the months in order *)
```

2.2 Real Types

VAX-11 PASCAL's predefined real data types allow you to express a wide range of real-number values with different degrees of precision. The identifiers REAL, SINGLE, DOUBLE, and QUADRUPLE denote the real types. REAL and SINGLE are synonymous; both denote single-precision real values. The type DOUBLE denotes double-precision real values. The type QUADRUPLE denotes quadruple-precision real values. In this manual, the term "real type" refers to the REAL, SINGLE, DOUBLE, and QUADRUPLE types collectively; the term "REAL type" refers to both the REAL and SINGLE types.

DOUBLE exists in two formats, G_floating and D_floating, which allow you to choose whether double-precision values will express a very wide range (G_floating) or a more limited range with somewhat greater precision (D_floating). You should not use both formats of DOUBLE in the same compilation unit; Section 10.6 describes how you can specify the double-precision format for a compilation unit by using an attribute.

Table 2-1 compares the range of values and the degree of precision for the real types.

Table 2-1: Range and Precision of Real Types

	SINGLE	D_FLOATING DOUBLE	G_FLOATING DOUBLE	QUADRUPLE
Smallest negative value	-0.29E-38	-0.29D-38	-0.56D-308	-0.84Q-4932
Largest negative value	-1.70E38	-1.70D38	-0.90D308	-0.59Q4932
Smallest positive value	0.29E-38	0.29D-38	0.56D-308	0.84Q-4932
Largest positive value	1.70E38	1.70D38	0.90D308	0.59Q4932
Precision	1 part in 2**23 = 7 decimal digits	1 part in 2**55 = 16 decimal digits	1 part in 2**52 = 15 decimal digits	1 part in 2**112 = 33 decimal digits

Real numbers can be written in either decimal or exponential notation. To write real numbers in decimal notation, you use the set of decimal digits and a decimal point. At least one digit must appear on either side of the decimal point. That is, a zero must always precede the decimal point of a real number between 1 and 0, and a zero must follow the decimal point of a whole number. The following are valid real numbers in decimal notation:

```
   2.4
 893.2497
   8.0
   0.0
```

Some numbers are too large or too small to be written conveniently in the above format; therefore, PASCAL provides exponential notation as a second way of writing real numbers. The parts of a real number written in exponential notation are: a real number or an integer, an upper- or lowercase letter to denote the type of precision, and an integer exponent with its minus sign or optional plus sign. For example:

```
 2.3e2
10.0E-1
 9.14159E0
```

The letter E after the value means that the value is to be multiplied by a power of 10 and indicates a single-precision real number. The integer following the E tells which power of 10 is to be used and can be positive or negative. Thus, the real number 237.0 can be represented in any of the following ways:

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

To indicate a double-precision real number, you must use exponential notation. Replace the letter E with the letter D (upper- or lowercase) to indicate the exponent. The following examples illustrate double-precision format:

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

Similarly, the letter Q (upper- or lowercase) in exponential notation designates a quadruple-precision value. For example:

```
0.1143503
3362Q2
0.11825q-4
```

Exponential notation is also called floating-point format because the position of the decimal point "floats" depending on the exponent following the letter.

You can use negative real numbers in decimal and exponential notations. However, a negative real number such as -4.5e+3 is not a constant, but is actually an expression consisting of the negation operator (-) and the real number 4.5e+3. The use of negative integers in complex expressions may not produce the results you expect; see Section 3.2.1 for more explanation. The input operations described in Chapter 8 allow you to supply a leading plus or minus sign with integer values; output operations, also described in Chapter 8, automatically supply leading minus signs with negative integer values.

2.3 Structured Types

In PASCAL, a structured type differs from an ordinal or a real type because it can contain more than one component at a time. Each component can be of an ordinal, real, structured, or pointer type. You can either access individual components of the type or process the entire structure.

The structured types are characterized by the type(s) of their components and by the manner in which the components are organized. VAX-11 PASCAL has five structured types, as described in the following sections: RECORD, ARRAY, VARYING OF CHAR, SET, and FILE.

For each structured type except FILE, you express a constant value of the type by forming a constructor. An array or record constructor must contain one constant value of the appropriate type for each component of the structure. You use constructors in the following ways in a PASCAL program:

- In a CONST section to define symbolic constants
- In a VAR section to initialize variables of structured types
- In an executable section to assign values to variables of structured types
- In an executable section to pass parameters to PASCAL routines

To save storage space, you can pack an object of any structured type except VARYING OF CHAR. Packed structures are generally stored in as few bits as possible. To create a packed structured type, specify the reserved word PACKED in front of the type definition.

2.3.1 RECORD Type

A record is a group of components called fields, which may be of different types and which may contain one or more data items. The record type definition specifies the name and type of each field.

Syntax

```
[PACKED]RECORD
field-list
END
```

where the syntax of a field-list is:

```
[ { {field-identifier},... : [attribute-list] type};... [;variant-clause] } [ ; ] ]
  { variant-clause [ ; ]                                                  }
```

field-identifier

The name of a field. Note that you can specify no field identifiers if you wish, thus making the field list empty.

attribute-list

One or more identifiers that provide additional information about the field(s) (see Chapter 10).

type

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

variant-clause

The variant part of the record (see Section 2.3.1.1).

To refer to a field within a record variable, you specify the name of the variable and the name of the field, separating them with a period. For instance, the field identifiers Team.Wins, Team.Losses, and Team.Percent could refer to three fields of a record variable named Team. You can use a field anywhere in a program that a variable of the field type is allowed.

The names of the fields must be unique within a record type but can be repeated in different record types. For instance, you can define the field Percent only once within a particular record type. Other record types, however, could also have fields called Percent. Because you must use the name of the record variable to refer to the field, no ambiguity results if fields in different record types have the same name.

A record type can include fields that are themselves records. In such a case, the name of the field includes the name of every record within which it is nested. For example:

```
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;
```

If you declare a variable Order of this type, you refer to its fields as Order.Part, Order.Received.Month, Order.Received.Day, Order.Received.Year, and Order.Inventory.

In a record constructor, constant values of the appropriate types are listed within parentheses in the same order as the corresponding fields appear in the record type definition. Constructors for nested records are enclosed in nested parentheses. A record constructor is usually preceded by the record type identifier. The type identifier is optional in the following cases:

- When the constructor is used to initialize a record variable
- When the record constructor is nested inside another constructor

If the record type in the previous example were named Order_Rec, you could write the following constructor for it:

```
Order_Rec (213, (Feb, 1, 1958), 7407)
```

The constructor specifies a constant value of the correct type for each field in the record and retains the same order as the field list. Note that because the record type Received is nested inside type Order_Rec, you need not specify the type identifier Received.

Two attributes, KEY and POS, can be applied only to record fields. The KEY attribute allows you to designate one or more fields as the key field(s) of an indexed file. The POS attribute allows you to position record fields relative to the beginning of the record. See Chapter 10 and the *VAX-11 PASCAL User's Guide* for more information on these attributes.

2.3.1.1 Record Type Examples

1.
```
RECORD
Year : INTEGER;
Gross : REAL;
Net : REAL;
Deductions : INTEGER;
Itemized : BOOLEAN;
END;
```

This example shows a record type with six fields. A possible constructor for this type is:

```
(1979, 10000.0, 8000.0, 1500, FALSE)
```

2.
```
RECORD
Person : Name;
Address : RECORD
          Number : INTEGER;
          Street, Town : Name;
          Zip : 0..99999;
          END;
Age : 0..150;
END;
```

This example shows one record nested within another. To write a constructor for the record type shown, you must enclose a constructor for the record Address within the constructor for the entire record. For example:

```
('Blaise Pascal       ', (1623, 'Pensees Street
 'Clermont Alaska     ', 91662), 39)
```

2.3.1.2 Records with Variants — A record can include one or more fields or groups of fields called variants, which can contain different types or amounts of data at different times during program execution. Thus, two variables of the same record type can represent different data. To specify a variant, you must include a variant clause as the last field in a record type definition.

Syntax

CASE [[tag-identifier :]][attribute-list] tag-type-identifier OF
{case-label-list : (field-list)};...

The tag field consists of the elements between the reserved words CASE and OF. The tag field is common to all variants in the record type. Its data type corresponds to the case label values and determines the current variant. As the syntax description illustrates, you can specify the tag field in two ways:

1. tag-identifier : [[attribute-list]] tag-type-identifier

   The tag identifier and tag type define the name and type of the tag field. The tag type identifier must denote an ordinal type. You refer to the tag field in the same way that you refer to any other field in the record—with the record.field-identifier syntax.

2. [[attribute-list]] tag-type-identifier

   In the second form, there is no tag identifier you can evaluate to determine the current variant; therefore, you must keep track of the current variant yourself. The tag type identifier must denote an ordinal type.

tag-identifier

The name of the tag field.

attribute-list

One or more identifiers that provide additional information about the variant (see Chapter 10).

tag-type-identifier

The type identifier for the tag field.

case-label-list

One or more constant values of the tag field type. There must be one label for each possible value in the tag type.

field-list

The names, types, and attributes of one or more fields. At the end of a field list, you can specify another variant clause. (See Section 2.3.1 for the syntax of a field list; note that the field list can be empty.)

You can refer only to the fields in the current variant. You may not change the variant while a reference exists to any field in the current variant. (The conditions that establish a variable reference are listed in Section 4.3.)

When you specify the tag field using a tag identifier, the current variant is the one whose label is equal to the current value of the tag identifier. Until you assign a new value to the tag identifier, you cannot refer to a field having a different case label. The following example shows the use of the tag identifier form:

```
RECORD
Part : 1..9999;
CASE Onorder : BOOLEAN OF
               TRUE  : (Order_Quantity : INTEGER;
                        Price: REAL);
               FALSE : (Rec_Quantity : INTEGER;
                        Cost : REAL);
END;
```

In this example, the last two fields in the record vary depending on whether the part is on order. Records for which the value of the tag identifier Onorder is TRUE will contain information about the current order; those for which it is FALSE, about the previous shipment.

The second way of specifying the tag field uses a tag type without a tag identifier. A reference to one field identifier causes the corresponding variant to become the current one; all other variants become undefined immediately. The following example shows the specification of a tag field without a tag identifier:

```
RECORD
Patient : Name;
Birthdate : Date;
Age : INTEGER;
CASE Sex OF
     Male : (Beard : BOOLEAN);
     Female : (Births : 1..30);
END;
```

In this example, assume that the tag field Sex is of an enumerated type with constant values Male and Female. The last field in this record is either the Boolean field Beard, if Male is the variant most recently referred to, or the integer subrange Births, if Female is the variant most recently referred to.

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

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

This record includes a variant field for each woman based on whether she has children. A second variant, which contains the number of children, is defined for women who have children.

A constructor for a record type that contains variants must include values for the tag field and the field identifiers in the corresponding variant. You must specify a value for the tag field, even if it has no tag identifier, to ensure that the correct variant is initialized. For example, consider the following record type named Call:

```
RECORD
Caller : Name;
Time : REAL;
Subj : (Work, Play, Sales, Chat);
CASE BOOLEAN OF
     TRUE : (Hour : INTEGER);
     FALSE : ();
END;
```

A constructor for this record type might look like this:

```
Call ('Washington', 10.30, Chat, TRUE, 12)
```

The constructor initializes the tag field with the Boolean value TRUE and the field identifier Hour with the integer value 12. Note that the tag field is initialized even though it does not have an identifier.

To initialize this record type with the value FALSE for the tag field, you could write the following record constructor:

```
('Washington', 10.30, Chat, FALSE)
```

This constructor specifies the same values as the previous one for all fields except the tag field. The tag field value is now last in the list because the FALSE case of the variant specifies no additional fields.

2.3.2 ARRAY Type

An array is a group of components of the same type that share a common identifier. The array type definition specifies the type of the components and the type of one or more indexes by which the components are accessed.

Syntax

[PACKED]ARRAY[{[attribute-list] index-type},...] OF
[attribute-list] component-type

attribute-list

One or more identifiers that provide additional information about the index type or the component type (see Chapter 10).

index-type

The type of the index, which can be any ordinal type.

component-type

The type of the array components, which can be any type.

The indexes of an array must be of an ordinal type. You usually cannot specify the type INTEGER as the index type because such an array would exceed the available memory space. To use integer values as indexes, you must specify an integer subrange. (An exception to this rule is the conformant array parameter; see Section 6.3.5.)

To refer to an array component, specify the name of an array variable, followed by an index value enclosed in brackets. For example, if you declare a variable Letters of type ARRAY[1..26] OF CHAR, you refer to its components as Letters[1], Letters[2], Letters[3], and so on, through Letters[26].

You can use an array component anywhere in a program that a variable of the component type is allowed. The only operation defined for the array as a whole, however, is the assignment (:=) operation (see Section 5.2).

In an array constructor, a constant value of the appropriate type for every component is listed within parentheses. To specify the same value for consecutive components, you can use a repetition factor of the form:

n OF value

The integer n denotes the number of consecutive components that are to receive the same value; n must be a constant expression of type INTEGER. [1] The value can be either a signed constant or another constructor of the component type.

As for records, an array constructor is usually preceded by the array type identifier. The type identifier is optional in the following cases:

- When the constructor is used to initialize an array variable
- When the array constructor is nested inside another constructor

For example, you could write the following constructor for an ARRAY [1..8] OF REAL whose type identifier is Result:

```
Result (1.318, 4.2029, 2 OF 3.68, 7.0, 3 OF 9.6445)
```

This constructor includes the repetition factor 2 OF 3.68, which specifies the value 3.68 for the third and fourth components, and the repetition factor 3 OF 9.6445, which specifies the value 9.6445 for the last three components.

2.3.2.1 Multidimensional Arrays — An array whose components are themselves arrays is multidimensional because it has more than one index. An array can have any number of dimensions, and each dimension can have a different index type. For example, the following syntax illustrates a two-dimensional array type:

```
ARRAY[0..4] OF ARRAY['A'..'D'] OF INTEGER
```

You can abbreviate the syntax by specifying all the index types in one pair of brackets. For example:

```
ARRAY[0..4, 'A'..'D'] OF INTEGER
```

To refer to a component of a two-dimensional array, specify the name of an array variable followed by two bracketed index values, written in the order in which their index types were declared. The first index indicates the rows of the array; the second index indicates the columns. For example, if you declare a variable Two_D of the array type shown above, you could refer to the components as Two_D[0,'A'], Two_D[0,'B'], and so on. You could also use the alternative form Two_D[0]['A']. Figure 2–1 represents the array variable Two_D.

1. In a constructor, the constant n cannot be a constant expression beginning with a parenthesis if the value being repeated is of type RECORD or ARRAY.

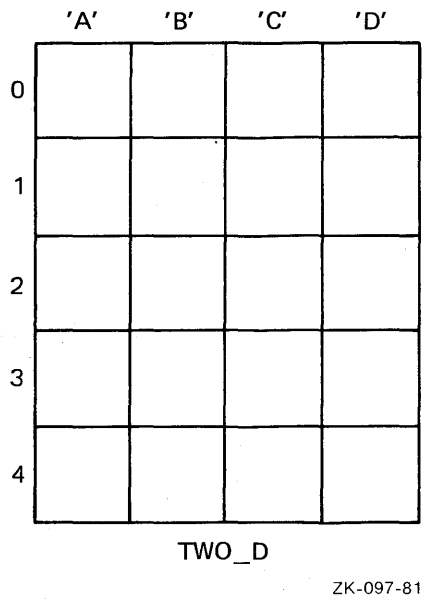
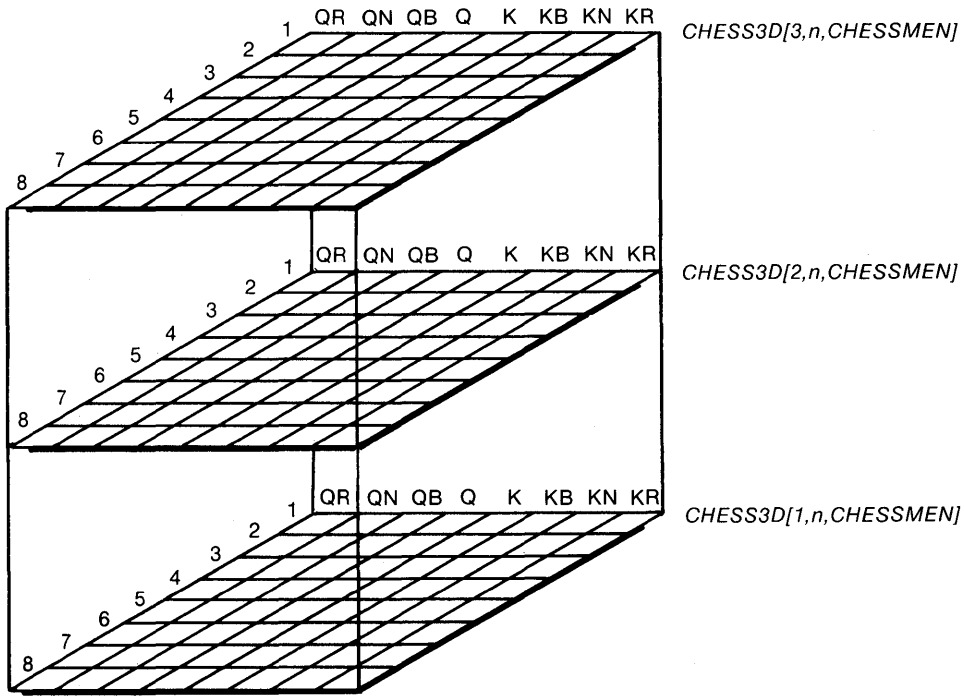


Figure 2–1: Two_Dimensional Array Two_D

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

If you do not specify a value for the rightmost index of a multidimensional array, you are referring to a component of an array type. For example, Two_D[0] refers to the entire first row of the array Two_D. This row is itself an array with four integer components.

You can construct arrays of three or more dimensions in a similar fashion. For example, suppose you create an enumerated type Chessmen with the values (QR,QN,QB,Q,K,KB,KN,KR,P,E). You could then declare a variable Chess3D of type ARRAY[1..3, 1..8, QR..KR] OF Chessmen. This array specifies a three-dimensional chessboard whose indexes represent the levels, ranks, and files of the chessboard. For example, the reference Chess3D[1] indicates one level, or a single chessboard. The reference Chess3D[1,1,QR] specifies the first level, first square in the upper left corner (bottom level, first rank, Queen's Rook file). Figure 2–2 illustrates the three levels of this array.



ZK-098-81

Figure 2–2: Three_Dimensional Array Chess3D

Just as a multidimensional array is really an array of arrays, so a constructor for a multidimensional array is a constructor whose components are constructors. You must include a constant value for each component of each array. For example, the syntax ARRAY[0..3, 1..5] OF REAL describes a two-dimensional array of real numbers. A constructor for an array of this type must consist of four constructors, each having five real values. One possible constructor is:

```
((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))
```

If you imagine the first index of this array as representing rows and the second index as representing columns, then the constructor above is filling the columns of the array one row at a time. Figure 2–3 shows the assignment of the above constructor to an array variable.

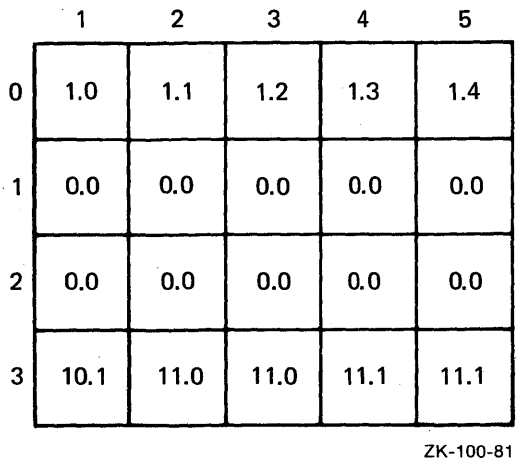


Figure 2-3: Values Assigned to a Two_Dimensional Array

You write a constructor for an array with three or more dimensions in a similar way. For example, for the array type

```
ARRAY[0..1] OF ARRAY[2..4] OF ARRAY[1..3] OF INTEGER
```

you could write the constructor

```
(((1,2,3), (20,40,60), (98,99,100)),
   ((5,7,9), (25,50,75), (100,200,300)))
```

For all but the innermost dimension, the constant values are actually written in the form of constructors because the component type of these arrays is another array. At the innermost dimension, the constant values are integers. Note that you must nest the constructors in the order in which the corresponding array types were defined.

2.3.2.2 Fixed-Length Character Strings — A fixed-length character string in PASCAL is defined as a packed array of characters with a lower bound of 1. The length of the string is established by the array's upper bound. The following example illustrates a fixed-length character-string type:

```
PACKED ARRAY[1..20] OF CHAR;
```

A variable of this type must contain a string of exactly 20 characters. The compiler will not add blanks to extend a shorter string, nor will it truncate a longer string. If you specify a string of incorrect length, an error occurs.

Note that if the upper bound of the array exceeds 65,535, the array is not considered to be a character string and cannot be treated as one in a program.

There are two ways to form a string constructor:

- Enclose in apostrophes a string of characters of the correct length
- Surround individual characters with apostrophes, separate them with commas, and enclose the list of characters in parentheses

With either method, you must provide one character constant for every component of the packed array. If the string does not have enough characters, you must add spaces to extend it. The following are valid constructors for a packed array of 10 characters:

```
'JEFFERSON '
('J','E','F','F','E','R','S','O','N',' ')
```

Some members of the ASCII character set, including the bell, the backspace, and the carriage return, are nonprinting characters. In VAX–11 PASCAL, you can include nonprinting characters within a character string.

Syntax

printing-string'({value},...) [['printing-string']]

printing-string

A character-string constant enclosed in apostrophes.

value

An integer denoting the ordinal value of an ASCII character.

You must close the string of printing characters with an apostrophe before you can indicate the nonprinting characters. After you have listed the ordinal values for the nonprinting characters, you can reopen the string and continue with printing characters. For example:

```
'A bell '(7)' in a null-terminated ASCII string'(0)
```

The ordinal value of the bell character is 7, and the value of the null character is 0. Note that the integers 7 and 0 are enclosed in parentheses within the character string.

The only nonprinting characters that can be inserted directly into a quoted string are the space and the tab.

2.3.2.3 Array Type Examples

1.
```
ARRAY[1..50] OF 0..200
```

   This example shows a 50-component array of integers in the subrange from 0 to 200. A constructor to give all the components the value zero might be:

   ```
   (50 OF 0)
   ```

2.
```
ARRAY[1..8, QR..KR] OF Chessmen
```

   This example shows a two-dimensional array that represents a chess board. Assume that the component type of the array, Chessmen, is the enumerated type (QR, QN, QB, Q, K, KB, KN, KR, P, E). You could write the following constructor to show how the chess pieces are arranged on the board at the start of a game:

   ```
   ((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))
   ```

   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. `PACKED ARRAY[1..10] OF CHAR;`

   For this array type, you could write the following string constructors:

```
'C.P.E.Bach'
'engrossing'
(10 OF ' ')
```

2.3.3 VARYING OF CHAR Type

The VARYING OF CHAR data type denotes a string of character components. The maximum length of the string is established by the VARYING OF CHAR type definition. Unlike a fixed-length packed array of characters, a VARYING string can have values of any length, from zero to the maximum specified.

Syntax

VARYING[upper-bound] OF [[attribute-list]] CHAR

upper-bound

An integer in the range from 1 through 65,535 that indicates the length of the longest possible string.

attribute-list

One or more identifiers that provide additional information about the VARYING string components (see Chapter 10).

When you declare a variable or component of type VARYING OF CHAR, the compiler allocates enough storage space to hold a string of the maximum length. The lengths of the character strings assigned to the variable or component may vary from zero to the specified maximum. A VARYING string with length zero is the empty string, `''`.

Although VARYING OF CHAR is a distinct type, it possesses some of the properties of both record and array types. A VARYING string is actually stored as though it were a record with two fields, LENGTH and BODY. The type syntax

VARYING[upper-bound] OF CHAR

may be thought of as the record type:

```
RECORD
LENGTH : [WORD] 0..upper-bound;
BODY : PACKED ARRAY[1..upper-bound] OF CHAR;
END;
```

LENGTH and BODY are predeclared field identifiers in VAX-11 PASCAL. The LENGTH field contains the length of the current character string; the BODY field contains the string. If your program requires it, you can access the values of LENGTH and BODY as you would access the values of record fields. Note that BODY is a fixed-length array large enough to contain a character string of the maximum length specified. (The WORD attribute is explained in Section 10.17.)

You can refer to the components of a VARYING string just as you refer to individual array components: by using the name of a VARYING string variable followed by an index value enclosed in brackets. For example, to access the fourteenth character of the variable Sentence, specify Sentence[14]. You may not specify an index value that is greater than the length of the current string. Enabling bounds checking causes this rule to be checked at run time (see Section 10.5 and the *VAX-11 PASCAL User's Guide*). Bounds checking is enabled by default.

The VARYING OF CHAR type does not have its own constructor syntax. Instead, it uses the same constructor syntax as a fixed-length character string, except that the length of the constructor can be shorter than the length specified in the type definition. When you need to assign to or initialize a variable of type VARYING OF CHAR, or when you need to pass a value to a formal parameter of type VARYING OF CHAR, you must use an expression that is assignment compatible with the variable or parameter (see Section 2.5).

Examples

1. `VARYING[25] OF CHAR`

   For this VARYING OF CHAR type, some possible values are:

   ```
   'Wolfgang Amadeus Mozart'
   'Bach'
   ```

2. `ARRAY[1..5] OF VARYING[20] OF CHAR`

   A constructor for this array type would have five string values, as in the following:

   ```
   ('Boston', 'Chicago', 'San Francisco',
    'Dallas', 'Minneapolis')
   ```

3. ```
   RECORD
   Title : VARYING[30] OF CHAR;
   Author : VARYING[20] OF CHAR;
   Category : (Fiction, Biography, Nonfiction, Children);
   END;
   ```

   A constructor for this record type must have two string values and a constant value of the enumerated type. For example:

   ```
   ('Gone with the Wind', 'Mitchell', Fiction)
   ```

2.3.4 SET Type

In PASCAL, a set is a collection of data items of the same ordinal type (called the base type). The set type definition specifies the values that can be elements of a variable of that type.

Syntax

[[PACKED]] SET OF [[attribute-list]] base-type

attribute-list

One or more identifiers that provide additional information about the base type (see Chapter 10).
   ```

base-type

The ordinal type identifier or type definition from which the set elements are selected. Note that real numbers cannot be elements of a set type.

You define a set by listing all the values that can be its elements. A set whose base type is INTEGER or UNSIGNED can have a maximum of 256 elements; the ordinal value of each element must be between 0 and 255. Therefore, integers outside the range of 0 through 255 cannot be set elements. For sets of other ordinal base types, elements can include the full range of the type.

To form a set constructor, enclose within brackets one or more constant values selected from the list of set elements. You can indicate consecutive values that appear in the set definition by using the subrange (..) symbol. For example, a constructor for a SET OF 35..115 could look like this:

```
[39, 67, 95, 110..115]
```

The set constructor contains nine values: 39, 67, 95, and all the integers between 110 and 115 inclusive.

A set having no elements is called an empty set and is written [ ].

Examples

1. `SET OF CHAR`

   Some possible constructors for this set type are:

   ```
   ['A', 'E', 'I', 'O', 'U']
   ['B'..'D', 'F'..'H', 'J'..'N', 'P'..'T', 'V'..'Z']
   ```

2. `SET OF 1..255`

   A constructor for a set of this type might include the following values:

   ```
   [3, 4, 15, 20, 23, 34, 40, 45, 55, 60, 70]
   ```

   Note that the upper limit of the subrange is the maximum allowed for a set of integers.

2.3.5 FILE Type

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

Syntax

[[PACKED]] FILE OF [[attribute-list]] component-type

attribute-list

One or more identifiers that provide additional information about the file components (see Chapter 10).

component-type

The type of the file components. It can be any ordinal, real, pointer, or structured type, except a file type or a structured type with a file component.

When you declare a file variable, the compiler automatically creates a file buffer variable of the component type; this variable takes on the value of one file component at a time. You can access only one file component, called the current component, at a given time. The predeclared input and output procedures described in Chapter 8 move the file position, thus changing the value of the file buffer variable. To denote the file buffer variable, write the name of the associated file variable and follow it with a circumflex (^). No operations may be performed on the file while a reference to the file buffer variable exists. (The conditions that establish a variable reference are listed in Section 4.3.)

For example, suppose you declare a file variable Math_Scores of type FILE OF INTEGER. As input and output procedures change the file position, the value of the file buffer variable Math_Scores^ also changes. Figure 2–4 shows the file positioned at the third component; the value of Math_Scores^ is 70.

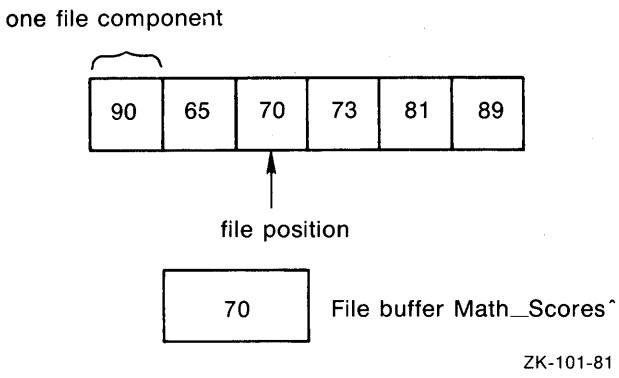


Figure 2–4: File Buffer Contents

The arithmetic, relational, Boolean, and assignment operators cannot be used with file variables or structures containing file components. You cannot form constructors of file types.

Examples

1.
```
FILE OF BOOLEAN
```

   This example shows a file of Boolean values. If you declare a variable Truthvals of this type, the file buffer variable is denoted by Truthvals^.

2.
```
FILE OF PACKED ARRAY[1..20] OF CHAR
```

   The components of this file type are strings of 20 characters. You could declare variables of this file type to contain lists of names, such as Accept_List, Reject_List, and Wait_List.

3.
```
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;
```

This example shows a file of records. If you declare a variable Results of this type, you would access fields of the record components as Results^.Trial, Results^.Date.Month, and so on.

2.3.5.1 External and Internal Files — A file that has a name in a directory and exists outside the context of a VAX-11 PASCAL program is known to VAX-11 Record Management Services (RMS) as an external file. A file that has no name and is not retained after the program finishes execution is known as an internal file. The OPEN procedure (see Section 8.3.1) creates an association between VAX-11 RMS and a file variable.

A file declared in the program heading is external by default. A file declared in a nested block is internal by default. You can change the default by giving an explicit name to an internal file. The file is then considered external and is retained with the specified name after the program has ceased execution.

2.3.5.2 Text Files — PASCAL supplies a predefined file type called TEXT. Variables of this type are called text files and have components of type CHAR. A text file differs from a file of type FILE OF CHAR in that it is divided into lines. Each line in a text file is a sequence of characters terminated by an end-of-line marker. You can refer to the marker indirectly through the predeclared procedures READLN and WRITELN (see Sections 8.7.4 and 8.7.5) and the predeclared function EOLN (see Section 8.7.1).

The predeclared file variables INPUT and OUTPUT are files of type TEXT. They refer to the standard input and output files, normally a terminal (in interactive mode) or the batch input and log file (in batch mode). These files are the defaults for all the predeclared text file procedures described in Chapter 8.

2.4 Pointer Types

Normally, variables exist as long as the program or routine in which they are declared is executing. By default, variables declared at program or module level are allocated in static storage; variables declared in nested blocks are allocated automatically on the stack. Some applications, however, require variables that have shorter or longer lifetimes within a program, or an unknown number of variables of a certain type. PASCAL allows you to use dynamic variables to fulfill these requirements.

Dynamic variables are allocated in an area called heap storage as they are needed during program execution. The NEW and DISPOSE procedures, described in Sections 7.5.2 and 7.5.3, allocate and deallocate dynamic variables.

Unlike other variables, dynamic variables do not have identifiers; you must refer to them indirectly with pointers. The pointer type definition identifies the type identifier of the dynamic variable.

Syntax

^[[attribute-list]] base-type-identifier

attribute-list

One or more identifiers that provide additional information about the base type (see Chapter 10).

base-type-identifier

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

A variable of a pointer type refers to a dynamic variable of the base type, and is said to be bound to that type. To indicate a pointer variable, write its name. To indicate the dynamic variable to which a pointer refers, write the name of the pointer variable followed by a circumflex (^). For example, suppose that M is a pointer variable bound to a record of type Myrec. Specify M^ to denote the record variable of type Myrec to which M refers.

Pointers assume values through initialization, assignment, the READ procedure (see Section 8.4.2), and the NEW procedure. The value of a pointer can be either the storage address of a dynamic variable or the predeclared identifier NIL. NIL indicates that the pointer does not currently refer to a dynamic variable.

A file referenced by a pointer is not closed until either execution of the program terminates or the dynamic variable is deallocated with the DISPOSE procedure. If you do not want the file to remain open throughout program execution, you must use the CLOSE procedure (see Section 8.3.2) to close it.

Example

```
RECORD
Name : VARYING[30] OF CHAR;
Class : (Standby, Coach, First);
Non_Smoking : BOOLEAN;
Flight_Number : INTEGER;
Destination : VARYING[5] OF CHAR;
Next_Passenger : ^Reservation;
END;
```

Suppose you define the record type shown here and give it the type identifier Reservation. The field Next_Passenger is defined as a pointer to the type Reservation. You could declare a variable Ticket of type Reservation; then, by manipulating the pointer variable Ticket.Next_Passenger, you could create a linked list of records.

2.5 Type Compatibility

The VAX-11 PASCAL compiler enforces two forms of type compatibility:

- Structural compatibility
- Assignment compatibility

Structural compatibility, described in Section 2.5.1 determines the types of data you can pass as VAR parameters and the types of pointer assignments you can make. Assignment compatibility, presented in Section 2.5.2, determines the types of values you can assign to variables of each type.

2.5.1 Structural Compatibility

Two types are structurally compatible only if they have the same allocation size and the same type structure. VAX-11 PASCAL requires that the type of a variable passed to a routine as an actual parameter be structurally compatible with the type of the corresponding formal VAR parameter. VAX-11 PASCAL also checks the structural compatibility of the base types when a pointer expression is assigned to a pointer variable.

Two ordinal types are structurally compatible only if they have the same base type and the same allocation size. The size may be established either by a size attribute (see Section 10.17) or by default. (See the *VAX-11 PASCAL User's Guide* for the default allocation sizes of ordinal types.)

If two ordinal types are components of packed structured types, they are structurally compatible only if the ranges of values they describe have identical upper and lower bounds.

In general, each real type is structurally compatible only with itself. However, because REAL and SINGLE are synonymous, they are structurally compatible with each other.

For two structured types to be structurally compatible, they must have the same allocation size and both must be packed or both unpacked. The following conditions also affect structural compatibility:

- If both types are record types, they must have the same number of fields, and the types of corresponding fields must be structurally compatible and identically positioned. If the record types have variant parts, the corresponding variants must have identical case labels written in the same order. The types of the fields within corresponding variants must be structurally compatible.
- If both types are array types, the types of their components must be structurally compatible. The index types must have identical base types and identical upper and lower bounds.
- If both types are VARYING OF CHAR types, their maximum lengths must be equal. The lengths of the current values of the VARYING strings do not affect structural compatibility.
- If two components of packed structured types are set types, their base types must have identical lower bounds and upper bounds.
- If both types are set types, file types, or pointer types, their base types must be structurally compatible. Because of the possibility that a pointer type can be defined in terms of itself, the VAX-11 PASCAL compiler begins the test for the structural compatibility of two pointer types by assuming that they are indeed compatible. Next, the compiler tests the two base types for structural compatibility. If within the base type, the compiler encounters the same pointer types it is testing, it still follows the original assumption that the pointer types are compatible. If the base types prove to be structurally compatible, then the two pointer types are in fact structurally compatible.

The effects of the alignment, POS, READONLY, UNSAFE, VOLATILE, and WRITEONLY attributes on structural compatibility are discussed in Chapter 10.

2.5.2 Assignment Compatibility

Assignment compatibility rules apply to the types of values used to initialize variables, the types of expressions assigned to variables using the assignment operator (:=), and the types of actual parameters passed to formal value parameters. Table 2–2 shows the contexts in which the type of an expression is assignment compatible with the type of a variable or a formal parameter.

Table 2–2: Assignment Compatibility

Type of Variable or Parameter	Type of Assignment-Compatible Expression
INTEGER	INTEGER
UNSIGNED	UNSIGNED, INTEGER
CHAR	CHAR
Subrange	Base type of the subrange
REAL, SINGLE	REAL, SINGLE, UNSIGNED, INTEGER
DOUBLE	DOUBLE, REAL, SINGLE, UNSIGNED, INTEGER
QUADRUPLE	QUADRUPLE, DOUBLE, REAL, SINGLE, UNSIGNED, INTEGER
PACKED ARRAY OF CHAR	CHAR, PACKED ARRAY OF CHAR with the same length, VARYING string whose current value is equal in length to the packed array
VARYING OF CHAR	CHAR, PACKED ARRAY OF CHAR, VARYING string whose current value does not exceed the maximum length of the variable or parameter
Pointer	Pointer to a structurally compatible type

Two record types or two array types are assignment compatible if they are structurally compatible. When you assign one record variable to another, or one array variable to another, the VAX-11 PASCAL compiler does not check for out-of-range assignments to record fields or array components; such assignments do not result in an error message, even if subrange checking is enabled at compile time. (See Section 10.5 and the *VAX-11 PASCAL User's Guide* for more information.)

A set expression is assignment compatible with a set variable if the sets' base types are compatible. In addition, all elements of the set expression must be included in the range of the variable's base type.

Note that assignment operations are not allowed on objects of file types or structured types that have file components.

The POS, READONLY, and UNSAFE attributes can change the rules of assignment compatibility; see Chapter 10 for complete descriptions of these changes.

Chapter 3
Expressions

An expression denotes a value. An expression may simply represent the value of a constant, a variable, or a function designator. Frequently, though, it involves the values of one or more such data items, or operands, combined with one or more operators.

VAX-11 PASCAL recognizes two forms of expressions: compile-time expressions and run-time expressions. A compile-time expression consists entirely of operands whose values can be determined when the program is compiled. The simplest compile-time expression is a single constant or constant identifier. Other compile-time expressions combine constants and constant identifiers with operators and the predeclared functions listed below (see Chapter 7 for complete descriptions):

- Arithmetic—ABS, ARCTAN, COS, EXP, LN, SIN, SQR, SQRT
- Ordinal—PRED, SUCC
- Boolean—ODD
- Transfer—CHR, DBLE, INT, ORD, QUAD, ROUND, SNGL, TRUNC, UINT, UROUND, UTRUNC
- Unsigned—UAND, UNOT, UOR, UXOR
- Allocation size—SIZE, NEXT, BITSIZE, BITNEXT
- Miscellaneous—CARD, EXPO

A run-time expression includes at least one operand whose value cannot be determined until the program is actually executed. A run-time expression contains one or more variables or function designators, but can also include constants, constant identifiers, and operators. You include a function designator within an expression by writing the function identifier and, optionally, a list of parameters that supply input values. The value of the function result is used in the expression. (See Chapter 6 for a complete discussion of writing function designators.)

When forming an expression, you are not limited to combining integers only with integers, real numbers only with real numbers, and so forth. PASCAL performs type conversions under certain circumstances, as described in Section 3.1, so that you can form expressions with operands of different types.

The operators used to form PASCAL expressions are the arithmetic, relational, logical, string, and set operators, all of which are explained in Sections 3.2.1 through 3.2.5. Although in general you cannot change the type of a variable once it has been declared, sometimes you might want to have this capability when forming expressions. Therefore, VAX-11 PASCAL allows you to alter temporarily the concept of a variable's type by using the type cast operator, as explained in Section 3.2.6. The order in which the operands in an expression are combined is determined by the precedence rules for the various operators, as described in Section 3.3.

3.1 Type Conversions

Since PASCAL is a strongly typed language, you cannot normally treat a value of one type as though it were of a different type, as you can in many languages. For example, you cannot assign the character '1' to a variable of type INTEGER, because '1' is not an integer constant but a character constant. However, there are times when it makes sense to combine values of two different types because the values have something in common. For example, suppose you wish to add a value of type REAL to a value of type INTEGER. This operation is legal because the value of type INTEGER is converted to its equivalent as a value of type REAL before the operation is performed. The result of the operation is of type REAL.

In PASCAL, values are converted from one type to another when the conversion is required for an operation, an assignment, or a formal/actual parameter association. Prior to any type conversion, the arithmetic types are ranked from lowest to highest:

INTEGER
UNSIGNED
REAL,SINGLE
DOUBLE
QUADRUPLE

Similarly, the character types are also ranked from lowest to highest:

CHAR
PACKED ARRAY OF CHAR
VARYING OF CHAR

When values of two different arithmetic or character types are combined in an expression, the lower-ranked operand is converted to its equivalent in the higher-ranked type. The result of an operation in which conversion occurs is always of the higher-ranked type.

Conversions to values of type UNSIGNED are never checked for overflow. When combined with other unsigned values, negative integer values are converted to large unsigned values by the calculation of the modulus with respect to 2**32 (see Section 3.2.1 for a description of the MOD operation).

A special case of conversion can occur when you attempt to assign an expression of type VARYING OF CHAR to a variable of type PACKED ARRAY OF CHAR. If the VARYING string has exactly the same number of components as the packed array, the VARYING string is converted to a packed array of characters before the assignment is made. If you attempt to perform this assignment with a VARYING string that has a different number of components than the packed array, a run-time error occurs.

3.2 Operators

PASCAL provides several classes of operators. You can form complex compile-time and run-time expressions by using operators to combine constants, constant identifiers, variables, and function designators.

PASCAL supplies the following classes of operators:

- Arithmetic operators
- Relational operators
- Logical operators
- String operators
- Set operators

3.2.1 Arithmetic Operators

An arithmetic operator usually provides a formula for calculating a value. To perform an arithmetic operation, you combine numeric data items with one or more of the operators listed in Table 3–1.

Table 3–1: Arithmetic Operators

Operator	Example	Result
+	A+B	Sum of A and B
-	A-B	B subtracted from A
*	A*B	Product of A and B
**	A**B	A raised to the power of B
/	A/B	A divided by B
DIV	A DIV B	Result of A divided by B, truncated toward zero
REM	A REM B	Remainder of A divided by B
MOD	A MOD B	Modulus of A with respect to B

Addition, subtraction, multiplication, and exponentiation (+, -, *, and **) operate on integer, unsigned, and real operands and produce a result of the same type as the values. If the expression contains operands of different types, PASCAL's conversion rules apply (see Section 3.1).

When you use a negative integer as an exponent, the exponentiation operation may yield unexpected results. The result of an integer raised to the power of a negative integer is defined as shown in Table 3–2.

Table 3–2: Results of Negative Exponents

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 integer, unsigned, and real operands, but always produces a real result. Use of the division (/) operator can therefore cause some loss of precision in expressions involving integer and unsigned operands.

DIV, REM, and MOD operate only on integer and unsigned operands. DIV divides one integer or unsigned operand by the other, producing an integer or unsigned result. DIV truncates toward zero any fraction; it does not round the result. For example, the expression 23 DIV 12 equals 1, and (-5) DIV 3 equals -1.

REM returns the remainder after dividing the first operand by the second. Thus, 5 REM 3 evaluates to 2. Similarly, 3 REM 3 evaluates to 0 and (-4) REM 3 evaluates to -1.

MOD returns the modulus of the first operand with respect to the second. The result of the operation A MOD B is defined only when B is a positive integer. This result is always an integer between 0 and B–1. The modulus of A with respect to B is computed as follows:

NOTE

The use of negative integer and real-number constants as operands in MOD and exponentiation operations may not produce the results you expect because the minus sign (-) is actually a negation operator. For example, the expression -2.0**2 is equivalent to the expression -(2.0**2) and produces the result -4.0. Therefore, you should enclose a negative constant in parentheses to make sure that it is interpreted as you intend. The expression (-2.0)**2 produces the result 4.0.

Table 3–3 lists the result types of arithmetic operations with operands of various types.

Table 3-3: Result Types of Arithmetic Operations

Operator	Type of Operands	Result Type
+ - * **	INTEGER, UNSIGNED, REAL, DOUBLE, QUADRUPLE	Same as the operands if both are of the same type; otherwise, the operand of the lower-ranked type is converted and the result is of the higher-ranked type
/	INTEGER, UNSIGNED, REAL, DOUBLE, QUADRUPLE	One of the real types — REAL if the operands are of type REAL (or SINGLE) or a lower-ranked type; otherwise, the operand of the lower-ranked type is converted and the result is of the higher-ranked type
DIV REM MOD	INTEGER and UNSIGNED only	INTEGER if both operands are of type INTEGER; UNSIGNED if the operands are of mixed types or are both UNSIGNED; otherwise, an error occurs

3.2.2 Relational Operators

A relational operator tests the relationship between two ordinal, real, string, or set expressions and returns a Boolean result. If the relationship holds, the result is TRUE; otherwise, the result is FALSE. Table 3-4 lists the relational operators that you can apply to arithmetic operands. You can also apply relational operators to string operands, as described in Table 3-6, and to set operators, as described in Table 3-7.

Table 3-4: Relational Operators

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

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

3.2.3 Logical Operators

A logical operator evaluates one or more Boolean expressions and returns a Boolean value. The logical operators are listed in Table 3-5.

Table 3-5: 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 value of a single condition so that if A is TRUE, NOT A is FALSE, and vice versa.

3.2.4 String Operators

A string operator concatenates or compares character-string expressions; its result is either a string or a Boolean value. The string operators are listed in Table 3-6.

Table 3-6: String Operators

Operator	Example	Result
+	A+B	String that is the concatenation of strings A and B
=	A=B	TRUE if strings A and B have equal ASCII values
<>	A<>B	TRUE if strings A and B have unequal ASCII values
<	A<B	TRUE if ASCII value of string A is less than that of string B
<=	A<=B	TRUE if ASCII value of string A is less than or equal to that of string B
>	A>B	TRUE if ASCII value of string A is greater than that of string B
>=	A>=B	TRUE if ASCII value of string A is greater than or equal to that of string B

With the plus sign (+), you can concatenate any combination of VARYING character strings, packed arrays of characters, and single characters.

The result of a string comparison depends on the ordinal value (in the ASCII character set) of the corresponding characters in the strings (see Appendix A). 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 TRUE because the digit 1 precedes the digit 2 in the ASCII character set.

The relational operators are legal only for character strings of the same length. The length of the current value of a VARYING string, not its maximum length, determines whether the string can be compared to a particular packed array of characters or another VARYING string. Enabling bounds checking causes the lengths of all character strings to be checked at run time for illegal operations (see Section 10.5 and the *VAX-11 PASCAL User's Guide*). Bounds checking is enabled by default.

3.2.5 Set Operators

A set operator forms the union, intersection, or difference of two sets, compares two sets, or tests an ordinal value for inclusion in a set. Its result is either a set or a Boolean value. Table 3–7 lists the set operators.

Table 3–7: Set Operators

Operator	Example	Result
+	A+B	Set that is the union of sets A and B
*	A*B	Set that is the intersection of sets A and B
-	A-B	Set of those elements of set A that are not also in set B
=	A=B	TRUE if set A is equal to set B
<>	A<>B	TRUE if set A is not equal to set B
<=	A<=B	TRUE if set A is a subset of set B
>=	A>=B	TRUE if set B is a subset of set A
IN	C IN B	TRUE if C is an element of set B

Most set operators require both operands to be set expressions. The IN operator, however, requires an ordinal expression as its first operand and a set expression as its second operand. The ordinal expression must be of the same type as the set's base type. For example:

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

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

The elements of a set constructor used in a set operation need not all be constants of the set type. Set elements are also allowed to be components of run-time expressions. For example, the set constructor

```
[i, j+5, k*1, m..q]
```

is a legal component of a run-time expression. If at run time, however, the value of m is greater than the value of q, the expression m..q would result in no set elements. In that case, the set constructor shown here would denote only three set elements.

3.2.6 Type Cast Operator

Every variable is associated with one and only one type: the type with which it was declared. Sometimes, however, you might be able to perform an operation more efficiently if you were able to relax temporarily PASCAL's strict type-checking rules.

VAX-11 PASCAL provides the type cast operator, which changes the context in which you can use a variable or an expression of a certain data type. The actual representation of the object being cast is never altered by the type cast operator. The type is simply overridden for the duration of one operation.

Syntax

variable-identifier :: type-identifier

or

(expression) :: type-identifier

The type cast operator (::) separates the name of a variable or an expression in parentheses from its target type, the type to which it is being cast. The operator "alters" the type of the cast object at that point only. The compiler assumes that a type cast will not affect the object at any other point in the program. Therefore, if the type cast is likely to affect the object elsewhere, the object should be declared with the VOLATILE attribute (see Section 10.21).

Once a variable or an expression has been cast, it has all the properties of its target type during the execution of the operation in which the type cast operator appears. A variable and its target type must have the same allocation size. Therefore, you may not cast a conformant variable parameter (see Section 6.3.5), although you may cast a fixed-size component of a conformant parameter.

When an expression in parentheses is cast, its value is either truncated on the left or padded on the left with zeros (if necessary) so that the allocation size of the expression's value and its target type become the same. The type of a cast expression cannot be VARYING OF CHAR (see Section 2.3.3) or a conformant schema (see Section 6.3.5). In addition, the target type of a cast expression cannot be VARYING OF CHAR. See the *VAX-11 PASCAL User's Guide* for the representation details for all the types.

Example

```
TYPE
   F_Float = PACKED RECORD
             Frac1 : 0..127;
             Expo : 0..255;
             Sign : BOOLEAN;
             Frac2 : 0..65535;
             END;

VAR
   A : REAL;
   .
   .
   .
A::F_Float.Expo := A::F_Float.Expo + 1;
```

In this example, the record type F_Float illustrates the layout of an F_ floating real number. The real variable A is cast as a record of this type, allowing you to access the fields containing the mantissa, exponent, sign, and fraction of A. Adding 1 to the field containing the exponent gives the same result as multiplying A by 2.0.

3.3 Precedence of Operators

The operators in an expression establish the order in which the operands are combined. Table 3–8 lists the order of precedence of the operators, from highest to lowest.

Table 3–8: Precedence of Operators

Operators	Precedence
::	Highest
NOT	
**	
*, /, DIV, REM, MOD, AND	
+, –, OR, Unary +, Unary –	
=, <>, <, <=, >, >=, IN	Lowest

In PASCAL, operators of equal precedence (such as + and –) are combined from left to right.

You must use parentheses for correct evaluation of an expression that combines relational operators. Consider, for example, the following expression:

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

Without parentheses, this expression would be interpreted as A<= (X AND B) <=Y and would result in an error if X and B are not of type BOOLEAN. The expression needs parentheses, as follows:

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

When the rewritten expression is evaluated, the Boolean values of the two relational expressions are combined with the AND operator.

You can use parentheses in an expression to force a particular order for combining the operands. For example:

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

The first expression is evaluated according to the normal precedence rules. First, 8 is multiplied by 5 and the result (40) is divided by 2. Then, 4 is subtracted to get 16. The parentheses in the second expression, however, force the subtraction of 4 from 2 (yielding –2) to be performed before the division of 40 by –2. The result is –20.

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.

The PASCAL compiler does not guarantee the order in which subexpressions, or the components of a complex expression, will be evaluated. In fact, some logical operations may be evaluated only partially if the result can be determined without complete evaluation. Usually the order of evaluation does not prevent the correct result from being produced. However, you should not overlook the importance of order in subexpression evaluation when you are writing logical operations involving function designators that have side effects. (A side effect is an assignment to a nonlocal variable or to a VAR parameter within a function block.)

For example, the following IF statement contains two function designators for function F:

```
IF F(A) AND F(B)
THEN
    .
    .
    .
```

Regardless of which function designator is evaluated first, if the result is FALSE, the other function designator does not have to be evaluated: the result of the IF test is likewise FALSE. Suppose that function F assigns the value of its parameter to a nonlocal variable. Because you cannot know which function designator was evaluated first, you cannot be sure of the value of the nonlocal variable after the IF statement is performed. Therefore, the desired results of your program should not depend on the order of subexpression evaluation.

Chapter 4
The Declaration Section

The first two parts of a PASCAL block are the heading and the declaration section. The heading specifies the name of the program, module, procedure, or function. The declaration section contains sections that define symbolic constants and user-created types, and sections that declare labels, variables, procedures, and functions. Each of these sections is introduced by an appropriate reserved word—LABEL, CONST, TYPE, VAR, PROCEDURE, or FUNCTION. A block need not include all of these sections. In VAX-11 PASCAL, those sections that are present may appear in any order and may appear more than once in a declaration section.

This chapter describes label declarations (Section 4.1), constant definitions (Section 4.2), type definitions (Section 4.3), and variable declarations (Section 4.4). Refer to Chapter 6 for information on procedure and function declarations.

4.1 Label Declarations

A label makes a statement accessible by a GOTO statement (see Section 5.7). A label is declared in a LABEL section; it is defined by its appearance preceding an executable statement. The declaration and the definition of a label must occur at the same level in the program.

Syntax

```
LABEL {label},...;
```

label

A decimal integer between 0 and MAXINT. When declaring several labels, 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 the label's declaration.

Example

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

This LABEL section specifies four labels: 0, 6656, 778, and 4352.

4.2 Constant Definitions

A CONST section defines symbolic constants by associating constant identifiers with compile-time expressions.

Syntax

CONST
    {constant-identifier = constant-expression};...

constant-identifier

The identifier of the symbolic constant being defined.

constant-expression

Any legal compile-time expression. As explained in Chapter 3, the VAX-11 PASCAL compiler must be able to evaluate all the components of a compile-time expression when it compiles the program.

Once a constant identifier is associated with an expression, it retains the value of that expression throughout program execution. You can change the value only by changing the definition in the CONST section.

You cannot access the individual components or fields of a symbolic constant that represents an array or record constructor.

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

Example

```
CONST
    Year = 1981;
    Month = 'January';
    Initial = 'P';
    Almost_Pi = 22.0/7.0;
    Tinyd = 1.7253D-10;
    Lie = FALSE;
    Untruth = Lie;
```

This CONST section defines seven symbolic constants. Year and Tinyd represent integer and double-precision numeric constants. Month represents a string constant, and Initial represents a character constant. The constant value of Almost_Pi is the real-number result of the expression 22.0/7.0. Both Lie and Untruth are equal to the Boolean value FALSE.

4.3 Type Definitions

A TYPE section introduces the name and set of values for a user-defined type.

Syntax

TYPE
    {type-identifier = [[attribute-list]]type};...

type-identifier

The identifier of the type being defined.

attribute-list

One or more identifiers that provide additional information for use when the type identifier appears in a declaration (see Chapter 10).

type

Any legal PASCAL type syntax.

PASCAL usually requires that a type identifier be defined before its subsequent use in the definitions of other types. In the only exception to this rule, PASCAL allows you to use a base type identifier in a pointer type definition before you define the base type. However, the base type must be defined before the end of the TYPE section in which it is first mentioned. 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 Ptr_to_Movie is defined as a pointer to the type Movie, which is defined later in the same TYPE section.

Example

```
TYPE
   Entertainment = (Dinner, Movie, Theater, Concert);
   Days_of_Week = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);
   Hours_Worked = ARRAY[Mon..Fri] OF INTEGER;
   Salary = ARRAY[1..50]OF REAL;
   Pay = Salary;
   Ptr_to_Hits = ^Hits;
   Hits = RECORD
          Title, Artist, Composer : VARYING[30] OF CHAR;
          Weeks_on_Chart : INTEGER;
          First_Version : BOOLEAN;
          END;
```

This TYPE section defines seven types and their identifiers. Both Entertainment and Days_of_Week are enumerated types. Hours_Worked is an array type with five integer components. Salary and Pay are identical array types of 50 real numbers each. Ptr_to_Hits is defined as a pointer to the type Hits, which is a record type having the five fields listed.

4.4 Variable Declarations

A VAR section declares variables and associates with each an identifier, a type, and possibly an initial value.

Syntax

VAR
{{variable-identifier},... : [[attribute-list]] type [[:= value]];...

variable-identifier

The identifier of the variable being declared.

attribute-list

One or more identifiers that provide additional information about the variable (see Chapter 10).

type

Any legal PASCAL type syntax.

value

Any assignment-compatible compile-time expression.

You can combine several identifiers in the same variable declaration if the variables are of the same type and are being initialized either with the same value or not at all. The following rules apply to variable initializations:

- Only statically allocated variables can be initialized. Variables declared at program or module level are statically allocated by default. To initialize a variable declared at an inner level, you must give it the STATIC attribute (see Section 10.3).
- You must initialize a variable with a compile-time expression of an assignment-compatible type. Scalar variables require scalar constants; structured variables require constant constructors.
- You cannot initialize file variables.
- The constant identifier NIL is the only value with which you can initialize a pointer variable.

A reference to a variable consists of the variable's use in one of the situations in the following list:

- The variable or one of its components is passed as a VAR (or %REF or %DESCR) parameter. The reference lasts throughout the call to the corresponding routine. (See Chapter 6 for a discussion of VAR, %REF, and %DESCR parameters.)
- The variable or one of its components is used on the left side of an assignment statement. The reference lasts throughout the execution of the statement. (See Section 5.2 for a discussion of the assignment statement.)
- The variable or one of its components is accessed by a WITH statement. The reference lasts throughout the execution of the statement. (See Section 5.6 for a discussion of the WITH statement.)

The existence of a variable reference sometimes prohibits certain operations from being performed on the variable. Such restrictions are noted in this manual.

Example

```
VAR
   Choice : Entertainment := Dinner;
   Answer, Rumor : BOOLEAN;
   Temp : INTEGER := 60;
   Grade : 'A'..'D';
   Next_Song : Ptr_to_Hits := NIL;
   Weekly_Hours : Hours_Worked := (7,8,7,9,6);
```

This VAR section declares seven variables and indicates the type of each. Choice is of the user-defined type Entertainment and is initialized with the constant identifier Dinner. Answer and Rumor are both Boolean variables. Temp is an integer variable initialized with the value 60. Grade is of a character subrange type consisting of the characters 'A', 'B', 'C', and 'D'. The pointer variable Next_Song is a pointer to the record type Hits defined in Section 4.3. Next_Song is given the constant identifier NIL as its initial value. The variable Weekly_Hours is declared to be of the user-defined array type Hours_Worked and is initialized with a constructor of integers.

Chapter 5
PASCAL Statements

PASCAL provides several statements that control the actions performed in a program. This chapter presents information, organized as follows, on each of these statements:

- The compound statement
- The assignment statement
- The empty statement
- Conditional statements:
  - CASE
  - IF-THEN
  - IF-THEN-ELSE
- Repetitive statements:
  - FOR
  - REPEAT
  - WHILE
- The WITH statement
- The GOTO statement
- The procedure call

PASCAL statements are classified as either simple or structured. The simple statements are the assignment, GOTO, and empty statements, and the procedure call. The structured statements are the compound, conditional, repetitive, and WITH statements. They enclose simple and structured statements that must be executed in order, repetitively, or when the specified conditions are met. You can use a structured statement anywhere in a block that a simple statement is allowed; therefore, this manual uses the term "statement" to mean either a simple or a structured statement.

5.1 The Compound Statement

The compound statement groups a series of statements so that they can be executed sequentially as though they were a single statement.

Syntax

```
BEGIN
{statement};...
END
```

statement

Any simple or structured statement.

A compound statement can combine any PASCAL statements, including other compound statements. The statements that make up the compound statement must be separated with semicolons. No semicolon is required between the last statement and the END delimiter; however, the examples in this manual show a semicolon before the END delimiter. This practice makes it easier to add new statements before the END at a later date.

Examples of compound statements appear throughout this chapter.

5.2 The Assignment Statement

The assignment statement assigns a value to a variable or function identifier.

Syntax

```
identifier := expression
```

identifier

The name of a function or any variable except a file variable.

expression

A run-time expression whose type is assignment compatible with the type of the variable.

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

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

You may not assign values to a variable of a record type with variants that was allocated with the NEW procedure (see Section 7.5.4); you may, however, assign values to a field of such a record variable.

Examples

1. `X := 1;`

   The variable X is assigned the value 1.

2. `T := A<B;`

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

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

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

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

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

5. `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.

5.3 The Empty Statement

The empty statement causes no other action to occur than the advancement of program flow to the next statement.

An empty statement can be represented by two consecutive semicolons. For example:

```
BEGIN
X := 10;
Y := 20;
Z[I] := 50;
;
END;
```

A common use of the empty statement appears in nested IF-THEN-ELSE statements (see Section 5.4.3).

5.4 Conditional Statements

A conditional statement causes a statement to be executed depending on the value of a controlling expression. PASCAL provides three conditional statements: CASE, IF-THEN, and IF-THEN-ELSE.

5.4.1 The CASE Statement

The CASE statement causes one of several statements to be executed, depending on the value of an ordinal expression called the case selector.

Syntax

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

case-selector

An expression of an ordinal type.

case-label-list

One or more constant values of the same ordinal type as the case selector, separated by commas.

Each case label corresponds to a statement that will be executed if the value of the case selector is equal to the case label. You can specify the case labels in any order; however, the difference in ordinal values between the largest and label and the smallest must not exceed 1000. Each case label can appear only once within a given CASE statement, but can appear in other CASE statements.

At run time, the system evaluates the case-selector expression and chooses which statement to execute. If the value of the case selector does not appear in the case label list, the system executes the statement(s) in the OTHERWISE clause. If you omit the OTHERWISE clause, the value of the case selector must be equal to one of the case labels.

Enabling case-selectors checking causes an error message to be produced at run time if the CASE statement fails to find an executable statement. When case-selectors checking is disabled, the result is undefined if the CASE selection fails and you have omitted an OTHERWISE clause. (See Section 10.5 and the *VAX-11 PASCAL User's Guide*.)

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 the case selector Age and executes the corresponding statement. The value of Age must be equal to 5, 6, 7, or 8.

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;
         Reading_Skill := FALSE;
END;
```

In this example, if the value of Age is not 5, 6, 7, or 8, the statements in the OTHERWISE clause are executed.

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

This example assigns the value of Vowel, Sometimes, or Consonant to Alpha_Flag, depending on the value of the case selector Alphabetic.

5.4.2 The IF-THEN Statement

The IF-THEN statement causes the execution of a statement, depending on the value of a Boolean expression.

Syntax

```
IF expression
THEN
     statement
```

expression

Any Boolean expression.

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

The THEN clause can specify either a simple or a structured statement. Note, however, that you must not place a semicolon between the word THEN and another statement (whether simple or structured). For example:

```
IF Day = Thurs
THEN;
(* misplaced semicolon *)
   BEGIN
      .
      .
      .
   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 the values of both relational expressions are TRUE, D is assigned the value of A–C. As discussed in Section 3.3, 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 the value of one of the relational expressions is FALSE, the other expression may not be evaluated.

3.
```
IF  (Name = 'Smith') AND (Initial = 'J')
THEN
   BEGIN
   Count := Count + 1;
   Smithadd[Count] := Address;
   END;
```

This example counts the number of people named J Smith and stores the street address of each person in an array.

5.4.3 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.

Syntax

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

expression

Any Boolean expression.

statement1

The statement to be executed if the value of the expression is TRUE.

statement2

The statement to be executed if the value of the expression is FALSE.

The object of a THEN or ELSE clause can be any simple or structured statement, including another IF-THEN-ELSE statement. For example:

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

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

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

D is assigned the value 1 if the values of both A and B are 1.

The ELSE clause always modifies the closest IF-THEN statement. Therefore, the object of the THEN clause in an IF-THEN-ELSE statement cannot be one of the following:

- An IF-THEN statement
- A structured statement ending with an IF-THEN statement

This restriction helps you avoid writing statements that may not execute as you had intended. For example:

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

Regardless of the format of this statement, PASCAL associates the ELSE clause with the statement IF B<>1 THEN C:=1. Thus, if the test IF A=1 is FALSE, no action is taken. To execute the ELSE clause when the test IF A=1 is FALSE, you could insert an empty statement, as follows:

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

Note that the object of the first ELSE clause is empty.

A semicolon preceding an ELSE clause terminates the IF-THEN-ELSE statement and causes a compile-time error. For example:

```
IF Age > Retire_Age
THEN
   Retired := TRUE; (* misplaced semicolon *)
ELSE
   Years_Left := Retire_Age - Age;
```

An error occurs when the reserved word ELSE is encountered because it is not a legal statement.

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.

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

5.5 Repetitive Statements

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

5.5.1 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.

Syntax

FOR control-variable := initial-value { TO / DOWNTO } final-value DO

statement

control-variable

The name of a previously declared variable of an ordinal type.

initial-value

An expression whose type is assignment compatible with the type of the control variable.

final-value

An expression whose type is assignment compatible with the type of the control variable.

The control variable, the initial value, and the final value must all be of the same ordinal type. The repeated statements, called the loop body, must not change the value of the control variable.

At run time, completion tests are performed before the FOR 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 body is executed and the value of the control variable is incremented. When the value of the control variable is greater than the final value, execution of the entire 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 body is executed and the value of the control variable is decremented. When the value of the control variable is less than the final value, execution of the entire loop is complete.

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

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

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

The value of the control variable is incremented or decremented in units of the appropriate type. For control variables of type INTEGER or UNSIGNED, one is added or subtracted to the value upon each iteration. For other types, the control variable takes on the successor (or predecessor) value of the type. For example, the value of a control variable of the subrange type 'A'..'Z' is incremented (or decremented) to the next character value each time the loop is executed.

If the FOR loop terminates normally (that is, if the loop exits because it is completed and not because of a GOTO statement), the value of the control variable is left undefined. You cannot assume that the control variable retains

a value. Therefore, you must assign a new value to the control variable before you use it elsewhere in the program.

If the FOR loop is terminated by a GOTO statement, the control variable retains the last value assigned to it and can be used outside the loop.

Examples

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

   This FOR loop computes the sum of the components 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 nest FOR loops. For each value of I, the executing program steps through all 10 values of the array J and assigns the value 0 to each component.

4.
```
FOR Employee := 1 TO N DO
   BEGIN
   Hrs := 0;
   FOR Day := Mon TO Fri DO
      IF NOT Sick[Employee,Day]
      THEN
         Hrs := Hrs + 8;
   Pay[Employee] := Wage[Employee] * Hrs;
   END;
```

   This example combines structured statements. The inner FOR statement computes the number of hours each employee worked from Monday through Friday. The outer FOR statement resets the number of hours to 0 for each employee and computes each person's pay as the product of Wage and Hrs.

5.5.2 The REPEAT Statement

The REPEAT statement executes one or more statements until a specified condition is true.

Syntax

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

expression

   Any Boolean expression.

The syntax of the REPEAT statement allows you to combine several statements between the reserved words REPEAT and UNTIL without BEGIN/END delimiters. The expression is evaluated after the statements are executed; therefore, the loop body is always executed at least once.

Example

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

Assume that the variable X is of type CHAR and the variables Digit_Count, Digit_Sum, and Char_Count denote integers. The example reads a character (X). If the value of X is a digit, the count of digits is incremented by one and the sum of digits is increased by the value of X, as computed by the ORD function. If the value of X is not a digit, the variable Char_Count is incremented by one. The REPEAT loop continues processing characters until it reaches an end-of-line condition.

5.5.3 The WHILE Statement

The WHILE statement executes one or more statements while a specified condition is true.

Syntax

```
WHILE expression DO
    statement
```

expression

Any Boolean expression.

The WHILE statement causes the statement following the word DO to be executed while the value of the conditional expression is TRUE. The expression is evaluated before the statement is executed. If the value of the expression is initially FALSE, the statement is never executed. The repeated statement must change the value of the expression; otherwise, the result is an infinite loop.

Unlike the REPEAT statement, the WHILE statement controls the execution of only one statement. Hence, to execute a group of statements repetitively, you must use a compound statement. Otherwise, only the single statement immediately following the word DO is repeated.

Examples

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

This statement skips to the end of 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. If the character is not a digit or letter, the error count (Err) is incremented by one.

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 0, the WHILE loop repeatedly calculates a student's average test score. If the average score falls below 90, the calculations cease and an informational message is printed. If the average never falls below 90, calculations continue until Ntests is greater than Maxtests; no message is printed.

5.6 The WITH Statement

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

Syntax

WITH {record-variable},... DO
    statement

record-variable

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

The WITH statement allows you to refer to the fields of a record by their names alone, rather than by the record.field-identifier syntax. In effect, the WITH statement opens the scope of the field identifiers so that references to field identifiers alone (not prefixed by the record name) are unambiguous.

Specifying more than one record variable has the same effect as nesting WITH statements. If the records themselves are nested, their names must appear in the order in which they were nested in the record type definition. If

the records are not nested, their names can appear in any order. Thus, the following two statements are equivalent:

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

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

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

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 Taxes.Itemized to TRUE if the value of Taxes.Net is less than 10000.0.

2.
```
TYPE
   Name = VARYING[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;
          END;
   .
   .
   .
WITH Hosp, Birthdate DO
   BEGIN
   Patient := 'Thomas Jefferson';
   Month := Apr;
   Day := 13;
   Year := 1743;
   END;
```

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

5.7 The GOTO Statement

The GOTO statement causes an unconditional branch to a statement prefixed by a label.

Syntax

```
GOTO label
```

label

An unsigned decimal integer that represents 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, including an empty statement.

The GOTO statement must be within the scope of the label declaration. A GOTO statement that is outside a structured statement cannot jump to a label within that structured statement. A GOTO statement within a routine can branch to a labeled statement in an enclosing block only if the labeled statement appears in the block's outermost level of nesting; that is, the labeled statement cannot occur within a structured statement.

Example

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

10: Invertsum := Result;
```

This example shows how you can use the GOTO statement to exit from a loop. The loop computes the sum of the inverses of the components of the variable Real_Array. If the value of one of the components is 0.0, however, the sum is set to 0.0 and the GOTO statement forces an exit from the loop.

5.8 The Procedure Call

A procedure call specifies the actual parameters to be passed to a procedure and executes the procedure.

Syntax

routine-identifier [({actual-parameter},...)]

routine-identifier

The name of a procedure or function.

actual-parameter

A run-time expression of an appropriate type, or the name of a procedure or function.

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. When the procedure has finished executing, control returns to the next executable statement following the procedure call.

The formal parameter list in the procedure declaration determines the possible contents of the actual parameter list. Depending on the types of the formal parameters, the actual parameters can be constants, variables, expressions, procedure identifiers, or function identifiers.

In VAX–11 PASCAL, a function may be called using the procedure call syntax. In this case, the value returned by the function is ignored. See Chapter 6 for a complete discussion of procedures and functions.

Examples

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

   This statement calls the procedure Tollbooth, and passes the variable Change, the real constant 0.25, and the first component of the array Lane as actual parameters.

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. `End_Process;`

   This statement calls the procedure End_Process, which has no parameters.

Chapter 6
Procedures and Functions

When designing a program that solves a complex problem, you may find it convenient to break down the problem into a collection of simpler subproblems. You can develop each subproblem independently and, once you have debugged it, you can be sure that it will execute successfully. In PASCAL, you can segment programs in this way by writing procedures and functions.

Procedures and functions, collectively called routines in this manual, have similar structures and restrictions. You can include routines in the main program, or you can compile them separately from the main program in modules. A VAX-11 PASCAL program can include user-written routines; external routines such as VAX/VMS system services, VAX-11 Run-Time Library routines, and routines written in other VAX-11 languages; and predeclared routines. External routines are discussed in greater detail in the *VAX-11 PASCAL User's Guide;* predeclared routines are described in Chapter 7.

This chapter is organized as follows:

- An overview of the concepts of PASCAL routines—Section 6.1
- The structure of a routine heading—Section 6.2
- The kinds of formal parameters—Section 6.3
- The properties of the routine block—Section 6.4
- The purposes of routine directives—Section 6.5
- The rules for the association of actual and formal parameters in routine calls—Section 6.6

6.1 Concepts of Routines

The overall algorithm for a program can usually be divided into relatively simple, repetitive tasks. In PASCAL, you can code each task separately as a routine; that is, as either a procedure or a function. Both procedures and functions associate a set of statements with an identifier; the statements are executed as a group when the routine is called from the executable section of the main program or another routine. In addition, a function returns a value of its declared type to the calling program or routine. You may call a function anywhere that an expression of its result type is allowed. Note that routines must usually be declared in a declaration section before they can be called.

A routine declaration consists of a heading and a body. The heading identifies the routine and may include a list of other identifiers, called formal parameters, if the routine needs to exchange data with the calling program or routine. The body is either a directive or a block. A directive supplies information about forward-declared and external routines; a block contains a declaration section (which may include nested routine declarations) and an executable section.

A routine exchanges data with the main program and with other routines by means of formal parameters and function results. The formal parameters used within the routine block must be listed in the routine heading. At run time, a formal parameter receives a value from an actual parameter in the routine call. You can call a routine several times with different actual parameters. The compiler checks every call to ensure that the types of the actual and formal parameters are compatible.

The scope of an identifier is the part of the program in which the identifier is accessible. In PASCAL, the scope of a label or an identifier (which represents a symbolic constant, variable, type, procedure, or function) is the block in which it is defined or declared, minus any nested blocks that redeclare the same label or identifier. The declaration section in the main program block introduces identifiers that are accessible in the main program and in all nested routines. The declaration sections in routine blocks specify local identifiers. You can use a local identifier in the routine that declares it and in all nested routines. In a routine, you can redeclare an identifier that has been declared in an outer block; the identifier always refers to the declaration of most limited scope.

6.2 Routine Headings

To declare a routine, you supply the routine's heading and either a block or a directive in a PROCEDURE or FUNCTION declaration section. Normally, you must declare a routine before you can call it from an executable section. The FORWARD directive, outlined in Section 6.5.1, allows you to escape this restriction.

The routine heading provides all the information necessary to determine whether the actual parameters in a call to the routine can legally be passed to the formal parameters in the declaration. Note that a procedure can have as

many as 255 formal parameters; certain functions, depending on their result types, are limited to 254 (see the *VAX-11 PASCAL User's Guide* for details).

Syntax

```
{ [[attribute-list]] PROCEDURE procedure-identifier [[formal-parameter-list]];

{ [[attribute-list]] FUNCTION function-identifier [[formal-parameter-list]]
                    : [[attribute-list ]] result-type-identifier;
```

attribute-list

One or more identifiers that provide additional information about the procedure, function, or function result (see Chapter 10).

procedure-identifier, function-identifier

The identifier that names the routine and, in the case of functions, also names the function result.

formal-parameter-list

The identifiers and types of the formal parameters and optionally the mechanism specifiers and attribute lists (see Section 6.3).

result-type-identifier

The type identifier of the function result, which can denote any type except a file type or a structured type with a file component.

A directive that may follow the heading declares that the routine is a FORWARD, FORTRAN, or EXTERNAL routine (see Section 6.5). A block that follows the heading contains an optional declaration section, which declares any data items that are local to the routine, and an executable section, which contains the statements that perform the routine's actions.

6.3 Formal Parameters

Formal parameters can be divided into three general categories: input parameters, output parameters, and routine parameters. A routine uses input parameters to obtain values; it uses output parameters to return values to the calling block. A function result is simply a special case of output parameter.

Some parameters act as both input and output parameters: the routine takes their values as inputs, modifies the values, and returns the changed values. In PASCAL, parameters used solely to supply input data are called value parameters (see Section 6.3.1); those used to return output values are called variable parameters (see Section 6.3.2).

Sometimes a routine requires the use of another procedure or function in order to perform its own actions. In PASCAL, a call to a routine can supply the name of another routine as a formal parameter (see Section 6.3.3).

When two routines exchange parameters, the calling routine must supply the data using the mechanism that the called routine expects. When declaring or calling routines not written in PASCAL, you may need to state an explicit mechanism for passing parameters. VAX-11 PASCAL includes a set of mechanism specifiers for declaring foreign (non-PASCAL) parameters (see

Section 6.3.4). Foreign parameters can be used as input, output, or routine parameters.

A formal parameter list may be composed of one or more of the five kinds of parameter sections listed below. A parameter section introduces one or more formal parameter identifiers and indicates how they will be interpreted within the routine.

- Value parameters—introduced without a reserved word
- Variable parameters—introduced by the reserved word VAR
- Procedure parameters—introduced by the reserved word PROCEDURE
- Function parameters—introduced by the reserved word FUNCTION
- Foreign parameters—introduced by a mechanism specifier (%REF, %IMMED, %DESCR, or %STDESCR)

The following sections describe the semantics of parameter passing in PASCAL and the use of each kind of parameter. Also described are conformant schemas (Section 6.3.5) and default parameters (Section 6.3.6).

6.3.1 Value Parameters

By the rules of value semantics, a formal value parameter represents a local variable within the called routine. The value of an actual parameter expression is passed to the called routine, which uses a copy of the value to initialize the formal parameter. The copy is not retained when control returns to the calling block. Therefore, if the called routine assigns a new value to the formal parameter, the change is not reflected in the calling block.

When you do not include a reserved word before the name of a formal parameter, you automatically cause PASCAL to use value semantics to pass data to that parameter.

Syntax

```
{identifier},... : [[attribute-list]] { type-identifier      }
                                      { conformant-schema    }
  [[:= [[mechanism-specifier]] default]];
```

identifier

The name of the formal parameter. Multiple identifiers must be separated with commas.

attribute-list

One or more identifiers that provide additional information about the formal parameter (see the text below and Chapter 10).

type-identifier

The type identifier of the parameters in this section.

conformant-schema

The type syntax of a conformant array or a conformant VARYING parameter (see Section 6.3.5).

mechanism-specifier

The mechanism by which a default value is to be associated with the formal parameter (see Sections 6.3.4 and 6.3.6).

default

A default value for the parameter (see Section 6.3.6).

Any attributes associated with a formal parameter become attributes of the local variable. They do not affect the values that can be passed to the parameter; they affect the behavior of the formal parameter only within the routine block. When a formal parameter has the UNSAFE attribute, the types of the actual parameters passed to it are not checked for compatibility (see Section 10.19).

The following are examples of formal value parameter sections in a routine heading:

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

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

6.3.2 Variable Parameters

By the rules of variable semantics, a formal variable parameter represents another name for a variable in the calling block. The routine directly accesses the actual parameter that corresponds to a formal variable parameter, rather than accessing a copy of it. Thus, the routine can assign a new value to the formal parameter during execution, and the changed value will be reflected immediately in the calling block.

PASCAL uses variable semantics to pass data to a formal parameter that is preceded by the reserved word VAR. Such a parameter is often called a formal VAR parameter.

Syntax

VAR {identifier},... : [[attribute-list]] { type-identifier / conformant-schema }
[[:= [[mechanism-specifier]] default]];

identifier

The name of the formal parameter. Multiple identifiers must be separated with commas.

attribute-list

One or more identifiers that provide additional information about the formal parameter (see Chapter 10 for details).

type-identifier

The type identifier of the parameters in this parameter section.

conformant-schema

The type syntax of a conformant array or a conformant VARYING parameter (see Section 6.3.5).

mechanism-specifier

The mechanism by which a default value is to be associated with the formal parameter (see Sections 6.3.4 and 6.3.6).

default

A default value for the parameter (see Section 6.3.6).

The following examples illustrate the formal VAR parameter sections of routine headings:

```
PROCEDURE Read_Write
   (VAR A : List);

FUNCTION Counter
   (VAR Instring, Outstring : VARYING[String_Size] OF CHAR;
   VAR Valid : BOOLEAN)
   : INTEGER;
```

Because no copy is made of the actual parameter, you can save storage space by using formal VAR parameters instead of value parameters. This technique can be especially helpful when you are passing actual parameters that require large amounts of storage. However, to use a VAR parameter as an efficient substitute for a value parameter:

- You must not modify the actual parameter.
- You should not refer to the actual parameter by more than one name within the same block.

The following example illustrates how passing a large array to a formal VAR parameter differs from passing it to a value parameter.

```
TYPE
   Big_Array = ARRAY[0..10000] OF REAL;

PROCEDURE Reverse
   (VAR Inarr, Outarr : Big_Array);

   VAR
      I, J : INTEGER;

   BEGIN
   J := 0;
   FOR I := 10000 DOWNTO 0 DO
      BEGIN
      Outarr[I] := Inarr[J];
      J := J + 1;
      END;
   END;

VAR
   A1, A2 : Big_Array;
      .
      .
      .
Reverse (A1, A2);                (* Would execute successfully *)
Reverse (A1, A1);                (* Would fail *)
```

The procedure Reverse is designed to reverse the order of the components of the array variable Inarr and write them to the array variable Outarr. You can save storage space by declaring Inarr and Outarr as formal VAR parameters, thus preventing the compiler from making copies of each 10,000-component array. The first call to Reverse illustrates this method.

In the second call to Reverse, however, the same array variable (A1) is passed to both Inarr and Outarr. Since Inarr and Outarr are formal VAR parameters, the procedure accesses the actual parameter A1 directly: Reverse actually modifies the input values as it writes the reversed components back into A1. Thus, the second call shown in this example would fail to execute as expected.

6.3.3 Formal Procedure and Function Parameters

Just as it is often convenient to subdivide a program into routines, it is often useful to break down routines even further into more procedures and functions. To declare a procedure or a function as a formal parameter to another routine, you must include a complete routine heading in the formal parameter list (see Section 6.2 for the syntax of a routine heading). You can associate a foreign mechanism specifier and a default value with a formal procedure or function parameter, as described in Section 6.3.6.

The following examples show formal procedure and function parameter sections in routine declarations:

```
PROCEDURE Apply
   (FUNCTION Operation (Left, Right : REAL) : REAL;
   Result : REAL);

FUNCTION Copy
   (PROCEDURE Get_Char (VAR C : CHAR);
   PROCEDURE Put_Char (I : CHAR))
   : BOOLEAN;
```

The identifiers listed as formal parameters to a formal procedure or function parameter are not accessible outside the routine declaration. They merely indicate the number and kind of actual parameters necessary. You refer to these identifiers only when you use nonpositional calling syntax to call a routine parameter. (Section 6.6.2 describes nonpositional syntax.)

In the above example, the formal parameter list of Get_Char informs the compiler that Copy must pass one character parameter to Get_Char using variable semantics. Copy does not refer explicitly to the formal parameter C unless it calls Get_Char using nonpositional syntax.

6.3.4 Foreign Mechanism Specifiers on Formal Parameters

When declaring PASCAL routines, you specify the semantics (value or variable) by which a formal parameter manipulates an actual parameter; the compiler is responsible for choosing the appropriate mechanism by which to pass the actual parameter. However, when declaring an external routine (one written in a language other than PASCAL) that is called by a PASCAL routine, you must specify not only the correct semantics but the correct mechanism as well.

VAX-11 PASCAL provides the foreign mechanism specifiers %IMMED, %REF, %DESCR, and %STDESCR, one of which can precede a formal parameter in the declaration of an external routine. If the formal parameter does not represent a routine, the mechanism specifier must precede the parameter name. If the formal parameter represents a routine, the specifier must precede the reserved word PROCEDURE or FUNCTION in the parameter declaration.

A mechanism specifier can also appear before the name of an actual parameter; see Section 6.6.7 for a description of this feature.

A mechanism specifier forces the use of mechanisms defined in the VAX-11 Procedure Calling Standard and also implies certain semantics. The passing of an expression to a foreign mechanism parameter implies foreign value semantics: the calling block makes a copy of the actual parameter's value and passes this copy to the called routine. The copy is not retained when control returns to the calling block. Note that foreign value semantics differs from value semantics in that the calling block, not the called routine, makes the copy.

The passing of a variable to a foreign mechanism parameter (except a parameter with the %IMMED specifier) implies foreign variable semantics: the variable itself is passed. A compile-time warning occurs if the compiler must convert the value of an actual parameter variable to make it match the type of a foreign mechanism parameter. In that case, the compiler passes a copy of the converted value by foreign value semantics using the specified mechanism. You can eliminate this warning by enclosing the actual parameter variable in parentheses; by doing so, you prevent the compiler from interpreting the actual parameter as a variable. The compiler takes the same action, whether or not it produces a warning message.

Mechanism specifiers on formal parameters produce the following results:

- A %REF formal parameter requires actual parameters to be passed using the by-reference mechanism. %REF implies variable semantics unless the actual parameter is an expression; in that case, it implies foreign value semantics.
- A %IMMED formal parameter requires actual parameters to be passed using the by-immediate-value mechanism and always implies value semantics. %IMMED cannot be used on formal parameters of type VARYING OF CHAR or on conformant array and conformant VARYING parameters (see Section 6.3.5).
- A %DESCR formal parameter requires actual parameters to be passed using the by-descriptor mechanism and interprets the semantics as %REF does.
- A %STDESCR formal parameter requires actual parameters to be passed using the by-string-descriptor mechanism. An actual parameter variable of type PACKED ARRAY OF CHAR implies variable semantics. An actual parameter expression of either type PACKED ARRAY OF CHAR or type VARYING OF CHAR implies foreign value semantics. You cannot use %STDESCR on formal procedure and function parameters.

Note that because the semantics is implicit in the mechanism, a formal parameter cannot be declared with both the reserved word VAR and a mechanism specifier.

As Section 6.6.7 describes, the VAX-11 PASCAL compiler checks for type compatibility when an external routine is called. However, at the time of the declaration, a %IMMED formal parameter that does not represent a routine is checked to ensure that it can be stored in 32 or fewer bits. A %IMMED formal parameter that does represent a routine must be declared with the UNBOUND attribute (see Section 10.18).

The *VAX-11 PASCAL User's Guide* provides further information about the use of foreign mechanism specifiers. Appendix C of the *VAX Architecture Handbook* defines the VAX-11 Procedure Calling Standard and explains descriptor formats.

6.3.5 Conformant Schemas

Some programming applications require general routines that can process arrays with potentially different bounds, or character strings with potentially different maximum lengths. Under PASCAL's rules of type checking, you would not easily be able to declare the type of such a parameter. Therefore, VAX-11 PASCAL provides conformant array and conformant VARYING schemas.

A conformant schema is a syntax that represents a set of types that are identical except for their bounds. The bounds of a conformant parameter are determined each time a corresponding actual parameter is passed. The bounds of an actual parameter are available within the routine through identifiers declared in the schema. A conformant schema can appear only within a formal parameter list.

You can use conformant schemas when declaring value, variable, and foreign mechanism parameters. When you use a conformant schema instead of a type identifier in a formal parameter declaration, a call to the routine can provide arrays and VARYING strings of different sizes, within the bounds specified by the schema. For example, you could write a procedure that finds the minimum, maximum, and average of the components of a one-dimensional array of integers. Similarly, you could write a function that returns the number of times one string occurs within another. On each call to such routines, the bounds of the formal parameters would be equal to those of the actual parameter.

Conformant Array Schema

```
{ ARRAY[{lower-bound-identifier..upper-bound-identifier:
        [[attribute-list]] index-type-identifier};...]
        OF [[attribute-list]] { type-identifier      }
                              { conformant-schema    }

{ PACKED ARRAY[lower-bound-identifier..upper-bound-identifier:
        [[attribute-list]] index-type-identifier]
        OF [[attribute-list]] type-identifier
```

lower-bound-identifier

An identifier that represents the lower bound of the conformant array's index.

upper-bound-identifier

An identifier that represents the upper bound of the conformant array's index.

attribute-list

One or more identifiers that provide additional information about the conformant array (see Chapter 10).

index-type-identifier

The type identifier of the index, which must denote an ordinal type.

type-identifier

The type identifier of the array components, which can denote any type.

Note that to specify the range and type of the index, you must use type identifiers that represent predefined or user-defined ordinal types. The identifiers that represent the index bounds can be thought of as READONLY value parameters, implicitly declared in the procedure declaration. Unless the conformant schema is packed, the component can be either a type identifier or another conformant schema; therefore, only the last dimension of a conformant schema can be packed.

Conformant VARYING Schema

[[attribute-list]] VARYING[upper-bound-identifier] OF CHAR

attribute-list

One or more identifiers that provide additional information about the conformant VARYING string (see Chapter 10).

upper-bound-identifier

An identifier that represents the upper bound of the conformant VARYING string's index.

The upper bound identifier specifies the maximum length of the VARYING string and must denote an integer. You can use the upper bound identifier in the body of the routine as a READONLY value parameter. The lower bound, which you do not declare, is always zero.

When you pass a conformant VARYING string expression to a value parameter, the length of the actual parameter's current value parameter (not its declared maximum length) becomes both the current length and the maximum length of the formal parameter. When you pass a conformant VARYING string variable to a VAR parameter, the declared maximum length of the actual parameter becomes the maximum length of the formal parameter.

Two conformant schemas (array or VARYING) are equivalent if they have indexes of the same ordinal type and components that either are structurally compatible or are themselves equivalent conformant schemas. They must also

have the same number of dimensions. Finally, either both must be packed or both unpacked.

Examples

1.
```
TYPE
   Workdays = 1..31;
   Feb_Days = 1..28;
   Mar_Days = 1..31;

PROCEDURE Inventory
   (VAR Amt_Sold : ARRAY[First_Day..Last_Day : Workdays]
      OF INTEGER);
```

The formal parameter Amt_Sold can have index values from 1 to 31 to indicate the number of workdays in each month. Thus, an actual parameter passed to Amt_Sold could be an array whose index type is either Feb_Days or Mar_Days. The procedure could then sum the components of Amt_Sold and return the monthly inventory total to the calling block.

2.
```
TYPE
   Level_Range = 1..6;
   Nclasses = 1..8;
   Nstudents = 1..40;
   Names = PACKED ARRAY[1..35] OF CHAR;

PROCEDURE Student_Count
   (School : ARRAY[Grade_Low..Grade_High : Level_Range;
                   Units_Low..Units_High : Nclasses;
                   Pupils_Min..Pupils_Max : Nstudents]
             OF Names);
```

This example declares School as a three-dimensional conformant array parameter. Note that it uses the abbreviated syntax for specifying the index type of a multidimensional array. Each array passed to School could contain the names of all the students in a particular elementary school. The indexes 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.

3.
```
PROCEDURE Dashed_Line
   (VAR String : VARYING[Len] OF CHAR);
```

In this example, note that Len is not a previously declared identifier but is instead an additional implicit parameter defined by the procedure declaration. The upper bound of the conformant parameter String is established by the declared maximum length of the actual parameter passed to it when the procedure Dashed_Line is called.

6.3.6 Default Formal Parameters

Sometimes when writing a routine, you can assume that every call to the routine will supply the same value for a particular parameter. If you were able to specify that value as a default for the formal parameter, then you would need to pass an actual parameter only if you wanted to supply a different value.

VAX–11 PASCAL allows you to associate a formal parameter with its default value when you declare it; to do so, you append the following information to the parameter declaration:

```
:= [[mechanism-specifier]] constant-expression;
```

The constant expression that follows the assignment operator (:=) is evaluated when the routine is declared. This default value, plus the optional mechanism specifier, must be a legal actual parameter for the kind of formal parameter with which the default is associated. The mechanism specifier is required when the formal parameter is a procedure or function so that type checking between the actual and formal parameters is suspended. Sections 6.6.4 through 6.6.7 provide the rules for writing actual value, variable, routine, and foreign mechanism parameters.

For example, suppose you declare the following routine:

```
FUNCTION Net_Pay
   (Hours : INTEGER;
   Tax : REAL := 0.05;
   Rate : REAL;
   Fica : REAL := 0.07;
   Overtime : INTEGER)
   : REAL;
```

The formal parameters Tax and Fica are given the default values 0.05 and 0.07, respectively. Unless a call to Net_Pay explicitly provides different values for these parameters, the defaults are used.

6.4 Blocks and Scope

As described in Section 6.1, a block contains a declaration section and an executable section. The declaration section declares labels and identifiers that are available within the block. An identifier declared in the declaration section can be used in subsequent declarations and definitions. The new labels and identifiers declared inside a block are local to that block and are unknown outside the scope of the routine.

By default, all local variables in routines are automatically allocated; that is, the system does not retain the values of local variables after it exits from the routine. Rather, each call to a routine creates copies of the local variables. Therefore, you can call a routine recursively without affecting the values held by the local variables at each activation of the routine. To preserve the value of a local variable (not the copy) from one call to the next, you must declare the local variable with the STATIC attribute (see Section 10.3).

The executable section of the block contains the statements that perform its actions. You can cause an exit from a block with one of the following statements: either the last executable statement of the block, which causes normal termination; or a GOTO statement, which transfers control to an outer block. You may not, however, use a GOTO statement in an outer block to transfer control into an inner block.

6.4.1 Scope of Identifiers

In PASCAL, the concept of scope is important because scope defines the legal limits of an identifier's accessibility. The scope of an identifier extends from its initial declaration to the end of the block, minus any nested blocks that redeclare the identifier. Scope rules help limit the declaration of an identifier to that part of the program in which the identifier is actually used. By taking advantage of scope rules, you can use an identifier more than once within a program and give it different meanings. You should, however, limit the redeclaration of identifiers to very short names, such as I, J, or X, to avoid confusion. The following rules of scope apply to PASCAL identifiers:

- An identifier can be declared only once within a particular scope.
- A previously declared identifier can be redeclared in a nested block.
- An identifier declared in the main program block is accessible in all nested blocks (except where it is redeclared); that is, its scope is the entire program.
- A procedure identifier can be redeclared within its own declaration section.
- A function identifier can also be redeclared, but not in a declaration section of the function's outermost block. Because a function identifier must have a value assigned to it, it can be redeclared only in a nested block.
- A formal parameter name follows the same rules of scope as a function identifier: it can be redeclared only in a nested block.
- A label declaration follows rules of scope similar to those for identifiers. The scope of a label is the block in which it is declared, minus any nested blocks that redeclare the label number. Therefore, you can transfer control from one block to an enclosing block, but you must follow certain restrictions, as outlined in Section 5.7.

Figure 6–1 illustrates the scope of identifiers that appear in several blocks in a program.

```
VAR
   A, B : INTEGER;
   .
   .
   .
PROCEDURE Level1a
   (Z, Y : INTEGER);

   TYPE
      C = ARRAY[1..35] OF CHAR;

   VAR
      D, E : C;
      .
      .
      .
   END; (* end PROCEDURE Level1a *)

PROCEDURE Level1b
   (V, U : CHAR;
   VAR T : INTEGER);
      .
      .
      .
   FUNCTION Level2: CHAR;

      VAR
         B : BOOLEAN;
         .
         .
         .
      END; (* end FUNCTION Level2 *)
      .
      .
      .
   END; (* end PROCEDURE Level1b *)
   .
   .
   .
```

Figure 6-1: Scope of Identifiers

Because of PASCAL's scope rules, the following statements about the identifiers declared in Figure 6–1 are true:

- Variable identifiers A and B are accessible everywhere in the example and, except in function Level2 (which redeclares B as the identifier of a BOOLEAN variable), they represent integers.
- Type identifier C and variable identifiers D and E are declared in procedure Level1a and are accessible in that block. However, the scope of C, D, and E

does not include those blocks that are outside the declaring procedure. You could not, for example, refer to the variable E in procedure Level1b because that block is outside the scope of the identifier E.

- Function Level2 redeclares the identifier B so that it represents a BOOLEAN variable rather than an integer. Inside Level2, B is BOOLEAN, but outside that block, B is still an integer. You may not redeclare B within the scope of the first block shown because B has already been declared there to denote an integer.
- The identifier Level1a is declared as a procedure identifier in the outermost block of the example. Level1a could have been redeclared in its own declaration section along with the procedure's local identifiers C, D, and E.
- The identifier Level2 is declared as a function identifier within procedure Level1b. Level2 cannot be redeclared within its own declaration section, but could be redeclared within a nested block.
- The formal parameter identifiers V, U, and T in procedure Level1b cannot be redeclared as local identifiers within that procedure, but could be redeclared within the nested block of function Level2.

6.4.2 Function Blocks

In PASCAL, a function identifier acts much like a variable and is synonymous with the function result. When the function is called, the value of its result is undefined. By the time the function has finished execution, a value whose type is assignment compatible with the result type must have been assigned to the function identifier. The last value assigned to the function identifier is the result that is returned to the calling block.

The function result may be of any ordinal, real, structured, or pointer type, except a file type or a structured type with a file component. Any attributes associated with the function result apply only within the function block. Assignment (:=) is the only operation allowed on the function result. You cannot pass a function identifier to a formal VAR parameter. You cannot access individual array components or record fields of the function result, nor can you access the storage to which a function result of a pointer type refers. A block may refer to a function identifier declared in an enclosing block, but only for the purposes of assigning a value to it and recursively calling it. If you use the function identifier as an expression within its own executable section, the result is a recursive call on the function.

6.4.3 Examples

The following examples show complete procedure and function declarations.

1.
```
PROCEDURE Min_Max_Avg
   (A : ARRAY[L..H : INTEGER] OF INTEGER;
   VAR Min, Max : Range;
   VAR Avg : REAL);

   VAR
      Sum, J : INTEGER;

   BEGIN
   Max := A[L];
   Min := Max;
   Sum := Max;
   FOR J := L+1 TO H DO
      BEGIN
      Sum := Sum + A[J];
      IF A[J] > Max
      THEN
         Max := A[J];
      IF A[J] < Min
      THEN
         Min := A[J];
      END;
   Avg := Sum/(H - L+1);
   END;
```

This procedure computes the minimum, maximum, and average values in array A. Min, Max, and Avg are formal VAR parameters whose values are returned to the calling block and can be used in other computations in the program. A is specified as a value parameter because the procedure is concerned only with the values in the array; the array is not an output parameter.

2.
```
FUNCTION Count_Substrs
   (VAR String : VARYING[Len1] OF CHAR;
   VAR KeyStr : VARYING[Len2] OF CHAR)
   : INTEGER;

(* This function returns the number of times one
   substring is found in another. *)

   LABEL
      10;

   VAR
      I, J, Count : INTEGER;
```

```
BEGIN
Count := 0;
FOR I := 0 TO String.Length - KeyStr.Length DO
   BEGIN
   FOR J := 1 TO KeyStr.Length DO
      IF String [I+J] <> KeyStr [J]
      THEN
         GOTO 10;
   Count := Count + 1;
   10:
   END;
Count_Substrs := Count;
END; (*Count_Substrs*)
```

The function Count_Substrs uses two formal VAR parameters, String and KeyStr. (Remember that you can access the LENGTH field of a VARYING string separately.) Count_Substrs returns an integer value that indicates the number of times KeyStr appears within String. Note that although formal VAR parameters are used here, the function does not modify them; they are used simply to save storage space.

6.5 Directives

A directive is the alternative to a block in a routine declaration. A directive provides the compiler with information about two kinds of routines: a routine whose heading is declared separately from its body, indicated by the FORWARD directive; and a routine that is external to the PASCAL program, indicated by the EXTERNAL (or, equivalently, the EXTERN or FORTRAN) directive.

To specify a directive, include it immediately after the routine heading and follow it with a semicolon (;). Directives are recognized only in this position in a routine declaration. When you use a directive, you must not follow the heading with a block. The following sections describe the two classes of directives.

6.5.1 FORWARD Declarations

Although PASCAL requires you to declare routines before you refer to them, a forward declaration allows a routine to refer to another routine whose block has not yet been specified. For example, if two routines call each other recursively, a complete declaration of both routines is impossible. Omitting the declaration is also impossible because without a formal parameter list, the routine cannot be compiled, nor can calls to the routine be verified. Therefore,

you must forward-declare one of the recursive routines. The forward declaration provides the compiler with the information it needs, just as any other declaration does. But the forward declaration allows you to withhold the specification of the routine block until later in the source file.

A forward declaration consists of the routine heading followed by the FORWARD directive, without a routine block. For example:

```
PROCEDURE Chestnut
   (Bld : REAL;
   Doc : CHAR;
   VAR Arc : Rec);
   FORWARD;
```

When you specify the block of a forward-declared routine, you supply only the appropriate reserved word (PROCEDURE or FUNCTION) and the routine name. You do not repeat the formal parameter list, the result type, and the attribute lists that may have appeared in the routine heading.

Example

```
[GLOBAL] FUNCTION Adder
   (Op1, Op2, Op3 : REAL)
   : REAL;
   FORWARD;

PROCEDURE Printer
   (Student : Name_Array);
      .
      .
      .
   BEGIN
      .
      .
      .
   Z := Adder (A, B, C);
      .
      .
      .
   END;

(* GLOBAL *) FUNCTION Adder;
   (* (Op1, Op2, Op3 : REAL) : REAL *)
      .
      .
      .
   BEGIN
      .
      .
      .
   Printer ('Leonardo da Vinci');
      .
      .
      .
   END;
```

This example forward-declares the function Adder, whose block appears after the declaration of the procedure Printer. Note that the heading of the Adder block describes its formal parameters, result type, and attribute list within

comment delimiters. Although you must omit the parameter list, result type, and attribute lists when you declare the function block, inserting this information as a comment is good documentation practice.

6.5.2 EXTERNAL Routines

The EXTERNAL, EXTERN, and FORTRAN directives indicate routines that are external to a PASCAL program. They are used to declare independently compiled PASCAL routines and routines written in other languages, including VAX/VMS system services and VAX–11 Run-Time Library routines. In VAX–11 PASCAL, the FORTRAN, EXTERN, and EXTERNAL directives are equivalent. However, to ensure the portability of your program, you should use the FORTRAN directive only for external routines written in FORTRAN.

If you declare independently compiled PASCAL routines with the GLOBAL attribute (see Section 10.20), their names must be unique. That is, no two PASCAL routines with the GLOBAL attribute can have the same name, even if they are declared in different scopes or different compilation units.

External routines not written in PASCAL are the only routines that can be declared using the %IMMED, %REF, %DESCR, and %STDESCR mechanism specifiers. See Section 6.3.4 and the *VAX–11 PASCAL User's Guide* for details.

Examples

1.
```
FUNCTION MTH$TANH
    (Angle : REAL)
    : REAL;
    EXTERN;
```

   This example declares MTH$TANH, a VAX–11 Run-Time Library procedure, as an external routine.

2.
```
PROCEDURE Forstring
    (%STDESCR S : PACKED ARRAY[A..B : INTEGER] OF CHAR);
    FORTRAN;
```

   This example declares the FORTRAN procedure Forstring. The formal parameter list specifies S as a conformant array parameter that is passed by string descriptor.

6.6 Routine Calls

A PASCAL routine is activated by either the execution of a procedure call or the evaluation of a function designator in an executable section. The syntax for invoking procedures and functions is identical:

Syntax

routine-identifier [[({actual-parameter},...)]]

routine-identifier

The name of the procedure or function.

actual-parameter

A run-time expression of an appropriate type, or the name of a procedure or function.

Actual parameters are required unless the routine has no formal parameter list or unless default values are being used for all the formal parameters.

Although procedure calls and function designators have the same syntax, the ways in which you use them within an executable section are different. A procedure call is a statement by itself. A function designator usually does not appear by itself; it is an expression whose result is used within an executable statement.

For example, you could invoke the procedure Yearly_Totals as

```
Yearly_Totals (Amount_Purchased, Amount_Sold, Amount_Discount);
```

while you might invoke the function Compute_Interest as

```
Earnings := Compute_Interest (Investment, 0.13, 5);
```

The procedure Yearly_Totals is executed for its effects; the function Compute_Interest is executed to compute a value that is then assigned to the variable Earnings.

For those instances when the function result is irrelevant, VAX-11 PASCAL allows you to call a function as though it were a procedure by using an executable statement.

The topics discussed in the following sections include:

- The calling of functions as procedures—Section 6.6.1
- The association of formal and actual parameters—Section 6.6.2
- The effect of default parameters on association—Section 6.6.3
- The specific rules for passing actual parameters to formal value, variable, and routine parameters—Sections 6.6.4 through 6.6.6
- The presence of foreign mechanism specifiers in actual parameter lists—Section 6.6.7

6.6.1 Calling Functions as Procedures

Sometimes you may want to perform the operations contained in a particular function, even though the result returned by the function is meaningless to the rest of your program. In VAX-11 PASCAL, you can use a procedure call statement to activate a function. In such cases, the function result is ignored. You may not, however, pass a function to a formal procedure parameter.

For example, given the function declaration

```
FUNCTION Buf_Put
   (Varying_Ptr : VARYING[Len] OF CHAR)
   : BOOLEAN;
```

you could write the following statements to call it:

```
IF Buf_Put (Ptr_Vary)
THEN
    .
    .
    .
Buf_Put (Ptr_Vary);
```

In the first call, the THEN clause is executed if the value of the function result is TRUE. The second call treats Buf_Put as a procedure and disregards the result.

6.6.2 Parameter Association

A routine call must pass exactly one actual parameter for each formal parameter. The actual parameter is either listed explicitly in the routine call or is supplied by means of a default value in the routine declaration.

One way of establishing the correspondence between actual and formal parameters is to give the parameters in each list the same position. That is, the association of actual and formal parameters proceeds from left to right, item by item, through both lists. This form of association is called positional syntax.

For example, suppose you declare the following procedure:

```
PROCEDURE Compute_Sum
   (X, Y : INTEGER;
   VAR Z : INTEGER);
```

Using positional syntax, you could issue the following procedure call:

```
Compute_Sum (Quantity + 6, 15, Total);
```

Thus, the formal parameter X is passed the value of Quantity + 6; Y is passed the integer value 15; and Z is passed the variable Total. Quantity and Total must be accessible as integer variables in the block from which Compute_Sum is called.

Another way of establishing correspondence is to specify the formal parameter name and the actual parameter being passed to it. In VAX-11 PASCAL, you can associate an actual with a formal parameter using the assignment (:=) operator. The actual parameters in the call do not have to appear in the same order as the formal parameters appeared in the declaration. This form of association is called nonpositional syntax.

Using nonpositional syntax, you could call the procedure Compute_Sum with the following statement:

```
Compute_Sum (Z := Total, X := Quantity + 6, Y := 15);
```

This call to Compute_Sum is equivalent to the call in the previous example that used positional syntax.

You may use both positional and nonpositional actual parameters in the same call. However, you must still supply at most one actual parameter for any

formal parameter, and you must list the positional parameters first. If you used both positional and nonpositional actual parameters in the same parameter list, the previous call to Compute_Sum might look like this:

```
Compute_Sum (Quantity + 6, Z := Total, Y := 15);
```

The first actual parameter, Quantity + 6, corresponds to the formal parameter X because both are the first parameters in their respective lists. Since the next two actual parameters use nonpositional syntax, they can be in either order, but they must be associated by name with the formal parameters to which they belong.

6.6.3 Default Parameters

When a routine call supplies no actual parameter for a formal parameter that was declared with a default value, the default is used. A compile-time error occurs if you fail to supply an actual parameter for a formal parameter that does not have a default value.

When you declare a formal parameter with a default value, you can either omit it from the routine call or, if you use positional syntax, (see Section 6.6.2) you can indicate its position with a comma. For example, consider the routine heading that was shown in Section 6.3.6:

```
FUNCTION Net_Pay
   (Hours : INTEGER;
   Tax : REAL := 0.05;
   Rate : REAL;
   Fica : REAL := 0.07;
   Overtime : INTEGER)
   : REAL;
```

You can call Net_Pay in one of two ways:

```
Take_Home_Year := Take_Home_Year +
                  Net_Pay (Overtime := Overtime_Week,
                           Rate := Pay_Rate,
                           Hours := Hours_Week);

Take_Home_Year := Take_Home_Year +
                  Net_Pay (Hours_Week, , Pay_Rate,
                           , Overtime_Week);
```

You can override a formal parameter's default value by associating the formal parameter with an actual parameter in a routine call. For example, if you wanted to replace the default value of the formal parameter Tax in the example above for one call, you could call Net_Pay as follows:

```
Take_Home_Year:= Take_Home_Year +
   Net_Pay (Hours_Week, 0.06, Pay_Rate, , Overtime_Week);
```

As a result of this routine call, the default value of Tax would be replaced by the value 0.06 supplied in the actual parameter list.

6.6.4 Actual Value Parameters

When a routine requires an actual parameter for input, you use value semantics to pass the actual parameter. An actual value parameter must be a compile-time or run-time expression whose type is assignment compatible

with the type of the corresponding formal parameter. Because there is no assignment compatibility for file variables, they can never be passed as value parameters.

If necessary, the type of an actual parameter is converted to the type of the formal parameter to which it is being passed. In this case, PASCAL follows the same type conversion rules that it uses to perform any other assignment (see Section 3.1). You may, for example, pass an integer expression to a formal parameter of a real type. If an actual parameter has the UNSAFE attribute, no conversion occurs (see Section 10.19).

When passing array and character-string expressions to conformant formal parameters, you must make sure that the components and indexes of both parameters are of the same base type. The index bounds of the actual parameter must fall within the range of the conformant array schema's index type. The rules for passing actual parameters to a conformant array parameter are affected by the UNSAFE attribute (see Section 10.19).

The following formal parameter list requires three value parameters:

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

You could write the following procedure call for the procedure Alpha:

```
Alpha (X+Y, 11, 'G');
```

if X and Y are integer variables. Note that the actual parameters corresponding to A and B must be integer expressions, and the actual parameter corresponding to C must be a character expression.

6.6.5 Actual Variable Parameters

When a routine requires an actual parameter as output, you use variable semantics to pass the actual parameter. Because the routine has direct access to the actual parameter, any change that the routine makes to the parameter's value is immediately reflected in the actual parameter.

In general, an actual VAR parameter must be a variable or a component of an unpacked structured variable; it cannot be any other expression unless the corresponding formal parameter has the READONLY attribute (see Section 10.16). You must pass a file variable to a formal VAR parameter. You may not pass the tag field of a variant record to a formal VAR parameter.

When passing array and character-string variables to conformant formal parameters, you must make sure that the components and indexes of both parameters are of the same base type. The index bounds of the actual parameter must be within the range of the conformant array schema's index type. The rules for passing actual parameters to a conformant array parameter are affected by the UNSAFE attribute (see Section 10.19).

The type of a variable passed to a routine must be structurally compatible with the type of the corresponding formal parameter. You cannot pass a component of a packed structure to a formal VAR parameter, although you can pass the entire structure.

The following formal parameter list contains three VAR parameters:

```
PROCEDURE Tempest
   (VAR Sea, Breeze : REAL;
   VAR Sick : Med_File);
```

You could call the procedure Tempest with this statement:

```
Tempest (Tide, Speed, Patient);
```

The actual parameters Tide and Speed must be variables of a real type. The actual parameter Patient must be a variable of the previously defined type Med_File.

In VAX-11 PASCAL, certain attributes in a routine declaration or a routine call affect the rules of compatibility between actual and formal VAR parameters. The resulting modifications to structural compatibility rules are outlined in Chapter 10. These rules also apply to the corresponding components of structured types and to the base types of pointer types used as formal parameters. The attributes that result in rule changes are the alignment, POS, READONLY, size, UNSAFE, VOLATILE, and WRITEONLY attributes.

6.6.6 Actual Procedure and Function Parameters

Sometimes a routine requires you to pass the name of a procedure or function as an actual parameter. When passing routines as parameters to other routines, VAX-11 PASCAL requires that the formal parameter lists in both declarations be congruent. As described in Section 6.3, a formal parameter list can have five different kinds of parameter sections: value, variable, procedure, function, and foreign mechanism. Two formal parameter lists are congruent if they have the same number of sections and if the sections in corresponding positions meet any of the following conditions:

- Both are value parameter sections containing the same number of parameters. The types of parameters must either be structurally compatible or be equivalent conformant schemas.
- Both are variable parameter sections containing the same number of parameters. The types of the parameters must either be structurally compatible or be equivalent conformant schemas. Any attributes associated with a formal VAR parameter affect the kinds of actual parameters that can be passed to it (see Section 6.6.5).
- Both are procedure parameter sections having either congruent formal parameter lists or no formal parameters.
- Both are function parameter sections having either congruent formal parameter lists or no formal parameters, and having structurally compatible result types.
- Both are foreign parameter sections having the same mechanism specifier and the same number of parameters, whose types must be structurally compatible.

If one formal parameter list has a LIST attribute on its last parameter section, the other formal parameter list must also have this attribute.

The following program shows a function declaration that includes two functions as formal parameters.

```
PROGRAM Money;

VAR
   Costs, Pay, Fedtax, Food : REAL;
   Housing : INTEGER;

FUNCTION Income
   (Salary,
   Tax : REAL)
   : REAL;
      .
      .
      .
FUNCTION Expenses
   (Rent : INTEGER;
   Grocery : REAL)
   : REAL;
      .
      .
      .
FUNCTION Budget
   (FUNCTION Credit (Earnings, UStax : REAL) : REAL;
   FUNCTION Debit (Housing : INTEGER; Eat : REAL) : REAL)
   : REAL;

   VAR
      Deduct : REAL;

   BEGIN (* FUNCTION Budget *)
      .
      .
      .
   Deduct := Debit (Eat := Food, Housing := Housing);
   Budget := Credit (Pay, Fedtax) - Deduct;
      .
      .
      .
BEGIN (* Main program *)
   .
   .
   .
Costs := Budget (Income, Expenses);
   .
   .
   .
END.
```

When the function Budget is called, the function Income is passed to the formal function parameter Credit, and the function Expenses is passed to the formal function parameter Debit. When Credit is called, the program-level variables Pay and Fedtax are substituted for Credit's formal parameters, Earnings and UStax. In the call to Debit, nonpositional syntax is used to associate Debit's formal parameters Housing and Eat with the program-level variable Housing and Food. Note that there is no conflict between the names of program-level variables and formal parameters of routine parameters.

The presence of the ASYNCHRONOUS and UNBOUND attributes in routine declarations causes additional requirements to be imposed on the routines that can legally be passed as actual parameters. See Chapter 10 for complete information about the effects of these attributes.

6.6.7 Foreign Mechanism Specifiers on Actual Parameters

When calling an external routine, you must make sure that you pass actual parameters by the mechanism stated or implied in the routine declaration. To this end, VAX-11 PASCAL allows you to use the foreign mechanism specifiers %IMMED, %REF, %DESCR, and %STDESCR before an actual parameter in a routine call.

When a mechanism specifier appears in a call, it overrides the type, semantics, and mechanism specified in the formal parameter declaration. Thus, type checking is suspended for the parameter association to which the specifier applies. (See the *VAX-11 PASCAL User's Guide* for more information on mechanism specifiers.)

Regardless of whether the mechanism is determined by a formal or an actual parameter, the mechanism specifier is interpreted in the same way (see Section 6.3.4).

Special considerations arise when a function that has no formal parameters of its own (or that has defaults that are being used for all its formal parameters) is passed as a formal parameter to another routine. The appearance of the function identifier in an actual parameter list could indicate the passing of either the address of the entry mask or the function result. In VAX-11 PASCAL, the address of the entry mask is passed by default. Therefore, to cause the function result to be passed, you must enclose the function identifier in parentheses.

For example, the following routine calls show the function F being passed by immediate value as an actual parameter to the procedure P:

```
P (%IMMED F);

P (%IMMED (F));
```

In the first example, the address of function F's entry mask is passed by immediate value to the procedure P. In the second example, function F is evaluated, and its result is then passed by immediate value to P.

Chapter 7
Predeclared Routines

VAX-11 PASCAL supplies predeclared procedures and functions that perform various commonly used operations. Note that predeclared functions always return a value that is associated with the function identifier. You use predeclared routines to:

- Perform arithmetic operations
- Return ordinal values
- Test Boolean relations
- Convert data from one type to another
- Create and destroy dynamic variables
- Pack and unpack array variables
- Perform operations on character strings and unsigned integers
- Determine the allocation size of a type
- Implement interlocked instructions
- Perform input and output (see Chapter 8)
- Perform other miscellaneous actions

In this chapter, the term "arithmetic types" refers to those data types that can be used in arithmetic operations. The arithmetic types are INTEGER, UNSIGNED, and the real types.

The following sections describe predeclared VAX-11 PASCAL routines in the order listed above. These routines are summarized in Appendix C.

7.1 Arithmetic Functions

Arithmetic functions perform mathematical computations. Actual parameters to these functions can be expressions of any arithmetic type. The predeclared arithmetic functions fall into two categories: fully generic functions and real generic functions.

7.1.1 Fully Generic Functions

Fully generic functions take an actual parameter of any arithmetic type and return a value of the same type. The fully generic functions are:

- ABS (x)—computes the absolute value of x.
- SQR (x)—computes the square of x.

7.1.2 Real Generic Functions

Real generic functions take an actual parameter of any arithmetic type and return a value of a real type. If the parameter is of type INTEGER, UNSIGNED, REAL, or SINGLE, the function returns a value of type REAL. If the parameter is of type DOUBLE or QUADRUPLE, the function returns a value of the same type. The real generic functions are:

- ARCTAN (x)—computes the arc tangent of x and expresses the result in radians.
- COS (x)—computes the cosine of x, which is expressed in radians.
- EXP (x)—computes the exponential of x; that is, e**x.
- LN (x)—computes the natural logarithm of x. The value of x must be greater than zero.
- SIN (x)—computes the sine of x, which is expressed in radians.
- SQRT (x)—computes the square root of x. If the value of x is less than zero, an error results.

7.2 Ordinal Functions

Ordinal functions require an actual parameter of an ordinal type and return a value of the same type. The ordinal functions are:

- PRED (x)—returns the value that immediately precedes x in the ordered sequence of values of its type. There must be a predecessor value for x in the type.
- SUCC (x)—returns the value that immediately succeeds x in the ordered sequence of values of its type. There must be a successor value for x in the type.

7.3 Boolean Functions

Boolean functions return one of the Boolean values FALSE and TRUE. In addition to the predeclared Boolean functions described here, VAX-11 PASCAL supplies the Boolean functions EOF, EOLN, and UFB, discussed in Chapter 8.

7.3.1 ODD (x)

The ODD function tests whether the value of x is odd. The parameter x must be of type INTEGER or UNSIGNED. The function returns TRUE if the value of x is odd and FALSE if the value of x is even.

7.3.2 UNDEFINED (r)

The UNDEFINED function tests whether r contains a reserved operand. The parameter r must be a variable of type REAL, SINGLE, DOUBLE, or QUADRUPLE. The function returns TRUE if r contains a value that has been reserved by VAX/VMS (see the *VAX Architecture Handbook* for details about VMS reserved values). If r does not contain a reserved value, the function returns FALSE. An error would result if you tried to use r in arithmetic computations.

7.4 Transfer Routines

Transfer routines take an actual parameter of one type and convert it to another type.

7.4.1 Transfer Functions

Transfer functions convert the value of an actual parameter to its equivalent in another type and return the converted value of the new type.

7.4.1.1 CHR (x) — The CHR function returns a value of type CHAR whose ordinal value in the ASCII character set is x, provided such a character exists. The parameter x must be of type INTEGER or UNSIGNED and have a value from 0 to 255.

7.4.1.2 DBLE (x) — The DBLE function converts the value of x to its double-precision equivalent and returns a value of type DOUBLE. The parameter x must be of an arithmetic type. The value of x must not be too large to be represented by a double-precision number.

7.4.1.3 INT (x) — The INT function converts the value of x to its integer equivalent and returns a value of type INTEGER. The parameter x must be of an ordinal type.

No error results if x is of type UNSIGNED and has a value greater than MAXINT. In that case, the value of x is converted to its equivalent as a 32-bit integer by subtracting 2**32 from it. For example, INT(3604928157) returns the value -690,039,139, which is the negative integer with the same 32-bit representation as the unsigned integer value 3,604,928,157.

7.4.1.4 ORD (x) — The ORD function returns as an integer the position of x in the ordered sequence of values of x's type. The parameter x must be of an ordinal type. Note that the ordinal value of an integer is the integer itself. If x is of type UNSIGNED, its value must not be greater than MAXINT.

7.4.1.5 QUAD (x) — The QUAD function converts the value of x to its quadruple-precision equivalent and returns a value of type QUADRUPLE. The parameter x must be of an arithmetic type.

7.4.1.6 ROUND (r) — The ROUND function converts the value of r to its integer equivalent by rounding the fractional part of the value. The parameter r must be of type REAL, SINGLE, DOUBLE, or QUADRUPLE. The value returned is of type INTEGER. The value of r must not be too large to be represented by an integer.

7.4.1.7 SNGL (x) — The SNGL function rounds the value of x to its single-precision equivalent and returns a value of type SINGLE. The parameter x must be of an arithmetic type. The value of x must not be too large to be represented by a single-precision number.

7.4.1.8 TRUNC (r) — The TRUNC function converts the value of r to its integer equivalent by truncating the fractional part of the value. The parameter r must be of type REAL, SINGLE, DOUBLE, or QUADRUPLE. The value returned is of type INTEGER. The value of r must not be too large to be represented by an integer.

7.4.1.9 UINT (x) — The UINT function converts the value of x to its equivalent as an unsigned integer and returns a value of type UNSIGNED. The parameter x must be of an ordinal type.

No error results if x is of type INTEGER and has a negative value. In that case, the internal representation of x is returned as an unsigned number.

7.4.1.10 UROUND (r) — The UROUND function converts the value of r to its equivalent as an unsigned integer by rounding the fractional part of the value. The parameter r must be of type REAL, SINGLE, DOUBLE, or QUADRUPLE. The value returned is of type UNSIGNED.

No error results if the value of r is negative or greater than 4,294,967,295. In that case, the unsigned result is the rounded parameter value MOD 4,294,967,296.

7.4.1.11 UTRUNC (r) — The UTRUNC function converts the value of r to its equivalent as an unsigned integer by truncating the fractional part of the value. The parameter r must be of type REAL, SINGLE, DOUBLE, or QUADRUPLE. The value returned is of type UNSIGNED.

No error results if the value of r is negative or greater than 4,294,967,295. In that case, the unsigned result is the truncated parameter value MOD 4,294,967,296.

7.4.2 Transfer Procedures

Transfer procedures pack and unpack array parameters.

7.4.2.1 PACK (a,i,z) — The PACK procedure copies components of an unpacked array variable to a packed array variable. PACK requires three parameters: an unpacked array variable a, a value i to indicate the starting value of a's index, and a packed array z of the same component type as a.

The number of components in a must be greater than or equal to the number of components in z. PACK (a,i,z) assigns the components of a, starting with a[i], to the array z, starting with z[low-bound], until all the components in z are filled.

In general, 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, where v is the upper bound of z and u is the lower bound of z. That is, ORD (n) must be greater than or equal to ORD (i) + ORD (v) - ORD (u).

Packing need not start with the first component of array a; for example, PACK (A,5,P) packs components A[5] through A[24] into components P[1] through P[20].

Examples

1.
```
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 components A[1] through A[20] to P[1] through P[20]; that is, all the components in A are packed into P.

2.
```
VAR
   A : ARRAY[1..25] OF 1..15;
   P : PACKED ARRAY[1..20] OF 1..15;
   .
   .
   .
PACK (A, 1, P);
```

This procedure moves components of array A into the packed array P. The parameter 1 specifies that the packing will start with array component A[1]. Thus, the components A[1] through A[20] are assigned to P[1] through P[20]. The components A[21] through A[25] are not moved.

7.4.2.2 UNPACK (z,a,i) — The UNPACK procedure copies components of a packed array variable to an unpacked array variable. The parameters required for UNPACK are identical to those required for PACK. The restrictions on the array indexes and the value of i are also the same as for PACK (see Section 7.4.2.1).

Normally, you cannot pass the individual components of a packed array to formal VAR parameters (see Section 6.6.5); you must first unpack the array.

Example

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

PROCEDURE Process_Components
   (VAR Ch : CHAR);
   .
   .
   .
READ (P);
UNPACK (P, A, 1);
FOR I := 1 TO 10 DO
   Process_Components (A[I]);
```

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

7.5 Dynamic Allocation Routines

VAX-11 PASCAL provides dynamic allocation routines for the creation of pointer variables. Using pointer variables and dynamic allocation routines, you can create linked data structures, as illustrated in Section 7.5.3.

7.5.1 ADDRESS (x)

The ADDRESS function returns a pointer value that refers to x. The parameter x must be a VOLATILE variable of any type except a component of a packed structured type. A compile-time warning results if x is a formal VAR parameter, a component of a formal VAR parameter, or a variable that does not have the VOLATILE attribute (see Section 10.21).

The VAX-11 PASCAL compiler assumes that all pointers refer either to dynamic variables allocated by the NEW procedure or to variables that have the VOLATILE attribute; a pointer cannot refer to a nonvolatile variable unless the variable is allocated in heap storage by the NEW procedure (see Section 7.5.2).

7.5.2 NEW (p)

The NEW procedure sets aside memory for p^, that is, the dynamic variable to which the pointer variable p refers. The value of this newly allocated variable (p^) is undefined. You cannot assume that the allocated area is initialized.

For example, you could declare a pointer variable as follows:

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

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

```
NEW (Ptr);
```

This procedure allocates a variable of type INTEGER in dynamically allocated heap storage. The variable is denoted by Ptr^, that is, the name of the pointer variable followed by a circumflex (^). This procedure also assigns the address of the allocated integer to Ptr.

7.5.3 DISPOSE (p)

The DISPOSE procedure deallocates memory for the dynamic variable p^. You refer to this variable using a pointer value.

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

```
DISPOSE (Ptr);
```

As a result, the memory allocated for Ptr^ is deallocated and the variable is destroyed. The value of the pointer Ptr becomes undefined. Because you cannot refer to an undefined quantity, you cannot call DISPOSE more than once for the same dynamic variable.

Example

```
PROGRAM Linked_List (INPUT, OUTPUT);

(* This program constructs a linked list of records.  Each
   student record contains the name and student ID number of one
   student and, in addition, 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 component at 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 begins by reading a number and a name for one student. The NEW procedure allocates memory for a new record named Student. This new record becomes the first record in the list; that is, Student^.Next points to the previous head of the list (or to NIL, if only one record has been read). The value of the new student record is assigned to the pointer variable Start.

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

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

The parameter pv must be a pointer variable of a type that refers to a record type with variants; the parameter p must be a pointer expression (including a pointer-valued function) of a type that refers to a record type with variants. In both cases, the optional t parameters must be constant expressions of an ordinal type. They represent nested tag field values, where t1 is the outermost variant.

If you create a dynamic variable without specifying the tag field values, enough memory is allocated 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 normally allocate, you can use the NEW (p,t1,...,tn) form. Because the record-with-variants form of the NEW procedure allocates memory for the variant alone and not for the whole record, you cannot assign or evaluate the record as a whole; you can assign and evaluate only the record's individual fields.

Example

```
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 as follows for only the variant that corresponds to Fish:

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

You can allocate memory for nested variants as follows:

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

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

The DISPOSE (pv,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.

If a dynamic variable with specified record variants was allocated by the NEW procedure, it can be deallocated only by a DISPOSE procedure that specifies identical record variants.

You may not dispose a dynamically allocated variable while a reference to it exists. Section 4.4 describes the conditions that establish a variable reference.

7.6 Character-String Routines

VAX-11 PASCAL supplies predeclared routines that manipulate character strings. The seven predeclared functions, BIN, HEX, INDEX, LENGTH, OCT, PAD, and SUBSTR, and the two predeclared procedures, READV and WRITEV, are described in the following sections.

7.6.1 BIN (x[, length[, digits]])

The BIN function converts the value of x to its binary equivalent and returns the binary digits in a string of type VARYING OF CHAR. The only parameter required is the expression to be converted; this parameter can be of any type except VARYING OF CHAR, a conformant array schema, or a conformant VARYING schema. Two optional integer parameters specify the length of the resulting string and the minimum number of significant digits to be returned. If you specify a length that is too short to hold the converted value, the resulting string is truncated on the left.

If you omit the optional parameters, the bit width of the converted parameter value determines the string length and the number of significant digits. By default, the number of significant digits is the minimum number of characters necessary to express all the bits of the converted parameter. This default length is one character more than the default number of digits, which causes a leading blank to be included in the resulting string when both parameters are omitted.

Example

```
TYPE
   Month_Dates = SET OF 0..31;

VAR
   Days_Of_Rain :  Month_Dates;
   .
   .
   .
Days_Of_Rain := [1, 2, 6, 10, 12, 14, 18, 22, 25, 30];
Result := BIN (Days_Of_Rain, 32);
```

In this example, the BIN function converts the value of Days_Of_Rain to its binary equivalent and returns this value as a string of 32 characters. The resulting string has a 1 in each position where a value was assigned to Days_Of_Rain and a 0 in all other positions. Thus, the string value returned by BIN for Days_Of_Rain is '01000010010001000101010001000110'. Note that the binary representation is from right to left, with the leftmost bit representing the set element 31.

7.6.2 HEX (x[, length[, digits]])

The HEX function converts x to its hexadecimal equivalent and returns the hexadecimal digits in a string of type VARYING OF CHAR. The parameters required for HEX and their default values are the same as those for BIN (see Section 7.6.1).

Example

```
VAR
   P : ^Rec;
   .
   .
   .
Digits := 8;
NEW (P);
Result := HEX (P, 10, Digits);
```

In this example, the HEX function returns a string of 10 characters containing the hexadecimal equivalent of the value of the pointer variable P. The string has 8 significant digits, as specified by the value of the actual parameter Digits.

7.6.3 INDEX (object, pattern)

The INDEX function locates the first occurrence of a pattern string within an object string. INDEX requires two character-string expressions as parameters: an object string to be searched and a pattern string to be found. The function returns an integer value that indicates the position where the leftmost component of the pattern string was located in the object string. The search ends as soon as the first occurrence of the pattern string is located. If the pattern string is not found, INDEX returns the value 0. If the pattern string is an empty string, INDEX returns the value 1. If the object string is an empty string, INDEX returns the value 0 unless the pattern string is also empty; in that case, INDEX returns the value 1.

Examples

1. ```
   Object_String := 'The Pilgrims landed at Plymouth Rock';
   Pattern_String := 'Plymouth Rock';
   Position := INDEX (Object_String, Pattern_String);
   ```

   The INDEX function searches the value of Object_String for the value of Pattern_String. The integer value returned in this example is 24, which indicates that the first character of Pattern_String occurs in position 24 of Object_String.

2. ```
   Object_String := 'The Pilgrims landed at Plymouth Rock';
   Pattern_String := 'Mayflower';
   Position := INDEX (Object_String, Pattern_String);
   ```

   The INDEX function searches the object string value 'The Pilgrims landed at Plymouth Rock', looking for the pattern string value 'Mayflower'. Since the function never finds Pattern_String within Object_String, it returns the integer value 0.

7.6.4 LENGTH (str)

The LENGTH function returns an integer value that indicates the length of a character-string expression that is its parameter.

Example

```
Current_String := 'Year-to-Date Sales';
Current_Length := LENGTH (Current_String);
```

The LENGTH function indicates the length of the current value of Current_String. Since this parameter has been assigned the value 'Year-to-Date Sales', the LENGTH function returns the integer value 18, indicating the number of characters in Current_String.

7.6.5 OCT (x[, length[, digits]])

The OCT function converts the value of x to its octal equivalent and returns the octal digits in a string of type VARYING OF CHAR. The parameters required for OCT and their default values are the same as those for BIN (see Section 7.6.1).

Examples

1. ```
   IntVar := 427;
   Result := OCT (IntVar, 10, 3);
   ```

   The OCT function returns the octal equivalent of IntVar in a string with 10 characters and 3 significant digits. The value returned in this example is '   653'. The string is padded on the left with enough blanks to extend it to the length specified.

2. ```
   Result := OCT (IntVar, 10, 10);
   ```

   If the value of IntVar is the same as in the previous example, the OCT function returns the value '0000000653'. The resulting string is padded with leading zeros to provide the 10 significant digits requested.

7.6.6 PAD (str, fill, size)

The PAD function appends a fill character to a character string as many times as is necessary to extend the string to its specified size. You must pass three parameters to PAD: a character-string expression to be padded, an expression of type CHAR to be used as the fill character, and an integer expression indicating the size of the final string. The function returns a character string of the desired size. This string is composed of the original string followed by the fill character, which is repeated as many times as is necessary to extend the string to its specified size.

The final size must be greater than or equal to the length of the string to be padded.

Examples

1.
```
Pad_String := 'Short string';
Result_String := PAD (Pad_String, '*', 20);
```

This example pads the value of Pad_String with the filler character '*' until the string is 20 characters long. Since Pad_String has the value 'Short string', the PAD function returns the character string 'Short string********'.

2.
```
Pad_String := 'Long character string';
String_Size := 10;
Result_String := PAD (Pad_String, '!', String_Size);
```

This example pads the value of Pad_String with the filler character '!' until the string is 10 characters long. Since Pad_String has been assigned the value 'Long character string', it already contains more than 10 characters. Therefore, an error occurs at run time.

7.6.7 SUBSTR (str, start, length)

The SUBSTR function extracts a substring from another character string. SUBSTR requires three parameters: a character string to be taken apart, an integer expression that indicates the starting position of the substring, and an integer expression that indicates the length of the substring. The function returns a character string of the length specified, starting at the specified position.

The following rules apply to the use of the SUBSTR function:

- The values of the starting position and the length must be greater than zero.
- There must be enough characters following the starting position to construct a substring of the specified length.

Examples

1. ```
   Original_String := 'This is the original string';
   Start_Position := 13;
   Substring_Length := 15;
   New_String := SUBSTR (Original_String, Start_Position,
                         Substring_Length);
   ```

   The SUBSTR function constructs a character string starting at position 13 of Original_String and containing the next 15 characters. It returns the character string 'original string'.

2. ```
   Original_String := 'The substring cannot be formed';
   New_String := SUBSTR (Original_String, 12, 25);
   ```

   In this example, an error at run time occurs because the SUBSTR function cannot construct a character string of length 25 beginning in position 12, because there are only 18 characters in Original_String following the specified starting position.

7.6.8 READV (str, parameter-list)

The READV procedure reads characters from a character-string expression and assigns them to the variables listed as parameters in the READV procedure call. The behavior of READV is analogous to that of READLN; the character string is analogous to a one-line file.

An error occurs at run time if values have not been assigned to all the parameters listed in the READV procedure call before the end of the character string is reached.

Examples

```
TYPE
   Color = (Yellow, Red, Blue);
   Flower = (Daisy, Rose, Orchid, Tulip);

VAR
   Paint : Color;
   Bouquet : Flower;
   Month : VARYING[9] OF CHAR;
   RealVar : REAL;
   Read_String : VARYING[17] OF CHAR;
   .
   .
   .
Read_String := 'Red July 26,33805';
```

1. ```
   READV (Read_String, Paint, Month, RealVar);
   ```

   The READV procedure reads characters from the string variable Read_String and assigns them to the variables Paint, Month, and RealVar.

2. ```
   READV (Read_String, Paint, Month, RealVar, Bouquet);
   ```

   In this example, when the READV procedure is called, the value of Read_String does not contain enough characters to assign values to all the variables listed. Therefore, an error occurs.

3. `READV (Read_String, Paint, Month);`

   In this example, characters are read from Read_String only until values are supplied for Paint and Month. The rest of the characters in the string are ignored.

4. `READV (Read_String, RealVar, Paint, Month);`

   In this example, the READV procedure tries to assign the first characters of Read_String to the variable RealVar. Because RealVar is of type REAL, the characters 'Red' cannot be assigned to it and an error occurs.

7.6.9 WRITEV (str, parameter-list)

The WRITEV procedure writes characters to a character-string variable of type VARYING OF CHAR by converting to textual representations the values of the parameters listed in the procedure call. The behavior of WRITEV is analogous to that of WRITELN; the character-string variable is analogous to a one-line file.

An error occurs if WRITEV reaches the maximum length of the character string before the values of all the parameters in the procedure call have been written into the string.

Example

```
TYPE
   Color = (Yellow, Red, Blue);
   Flower = (Daisy, Rose, Orchid, Tulip);

VAR
   Bouquet : Flower := Orchid;
   Month : VARYING[9] OF CHAR;
   RealVar : REAL;
   Write_String : VARYING[30] OF CHAR;
   .
   .
   .
RealVar := 232.705;
WRITEV (Write_String, Yellow, RealVar:7:3, PRED(Bouquet));
```

The WRITEV procedure writes the constant value Yellow, the value of RealVar with a specified field width (see Section 8.7.6), and the predecessor of the value of Bouquet into the variable Write_String. Write_String then contains the value

```
' YELLOW232.705   ROSE'
```

7.7 Unsigned Functions

VAX-11 PASCAL supplies the predeclared functions UAND, UNOT, UOR, and UXOR to perform binary logical operations on expressions of type UNSIGNED and return unsigned values. The operations performed by the functions are as follows:

- UAND (u1, u2)—performs a binary logical AND on the corresponding bits of the two expressions
- UNOT (u1)—performs a binary logical NOT on each bit of the expression
- UOR (u1, u2)—performs a binary logical OR on the corresponding bits of the two expressions
- UXOR (u1, u2)—performs a binary logical exclusive OR on the corresponding bits of the two expressions

Examples

1. ```
   Result := UAND (%X'FF9', %X'703');
   ```

   The UAND function performs a binary logical AND operation on each pair of bits and returns the unsigned hexadecimal value %X'701'.

2. ```
   Result := UNOT (%X'FF9');
   ```

   The UNOT function performs a binary logical NOT operation on each bit and returns the unsigned hexadecimal value %X'FFFFF006'.

3. ```
   Result := UOR (%X'FF9', %X'703');
   ```

   The UOR function performs a binary logical OR operation on each pair of bits and returns the unsigned hexadecimal value %X'FFB'.

4. ```
   Result := UXOR (%X'FF9', %X'703');
   ```

   The UXOR function performs a binary logical exclusive OR operation on each pair of bits and returns the unsigned value %X'8FA'.

7.8 Allocation Size Functions

VAX-11 PASCAL's allocation size functions provide information about the amount of storage allocated for variables and components of various types (see the *VAX-11 PASCAL User's Guide* for the default allocation size for items of each type). The parameters may be in the form of variable or type identifiers. Each function returns an integer value that represents the allocation size of the given parameter.

7.8.1 SIZE (x[,t1,...,tn])

The SIZE function returns an integer value that indicates the number of bytes that would be allocated for a variable or record field of type x.

The parameter to the SIZE function may be a variant record variable or type identifier. In that case, you can supply additional parameters t1 through tn that correspond to the case labels of the record. The SIZE function returns an integer value that indicates the number of bytes that would be allocated by the NEW procedure for a dynamic variable of the specified variant.

7.8.2 NEXT (x)

The NEXT function returns an integer value that indicates the number of bytes that would be allocated for one component of type x in an unpacked array.

A warning occurs if x represents a formal parameter because the alignment of the corresponding actual parameter cannot be determined. The formal and actual parameters are assumed to have the same alignment, but in fact, the actual parameter is allowed to have greater alignment.

Note that the NEXT and SIZE functions return the same byte size values for a given type, except when the components of the specified type in an unpacked array would have been padded to ensure proper alignment.

7.8.3 BITSIZE (x)

The BITSIZE function returns an integer value that indicates the number of bits that would be allocated for one field of type x in a packed record.

7.8.4 BITNEXT (x)

The BITNEXT function returns an integer value that indicates the number of bits that would be allocated for one component of type x in a packed array.

7.9 Low-Level Interlocked Functions

VAX-11 PASCAL provides low-level interlocked functions to allow parallel processes and asynchronous routines to operate in a real-time or multitasking environment. The compiler translates these functions into the interlocked machine instructions provided by the VAX-11 architecture.

7.9.1 ADD_INTERLOCKED (e, v)

The ADD_INTERLOCKED function adds the value of the expression e to the value of the variable v, using the VAX-11 Add Aligned Word Interlocked (ADAWI) instruction, and stores the newly computed value in v. The type of v must be an integer or an unsigned subrange; v must have an allocation size of two bytes and must be aligned on a word boundary. The type of e must be assignment compatible with that of v. The function returns the integer value -1 if the new value of v is negative, 0 if it is zero, and +1 if it is positive.

Note that unless the type of v is an integer subrange that includes negative values, the result of the ADD_INTERLOCKED function will never be -1.

Overflow and subrange checking are never performed on the ADD_INTERLOCKED operation, even if these options are in effect for the rest of the routine or compilation unit. (See Section 10.5 and the *VAX-11 PASCAL User's Guide* for details on checking options.)

7.9.2 CLEAR_INTERLOCKED (b)

The CLEAR_INTERLOCKED function assigns the value FALSE to b and returns the original value of b, using the VAX-11 Branch on Bit Clear and Clear Interlocked (BBCCI) instruction. The parameter b must be a variable of type BOOLEAN. The variable does not have to be aligned; therefore, it can be a field of a packed record.

7.9.3 SET_INTERLOCKED (b)

The SET_INTERLOCKED function assigns the value TRUE to b and returns the original value of b, using the VAX-11 Branch on Bit Set and Set Interlocked (BBSSI) instruction. The parameter b must be a variable of type BOOLEAN. The variable does not have to be aligned; therefore, it can be a field of a packed record.

7.10 Miscellaneous Routines

VAX-11 PASCAL supplies predeclared routines that determine the amount of time a process uses, record the system date and time, control error handling of a program, and perform miscellaneous calculations.

7.10.1 CARD (s)

The CARD function returns an integer value indicating the number of components that are currently elements of the set expression s.

7.10.2 CLOCK

The CLOCK function returns an integer value indicating the amount of central processor time in milliseconds used by the current process. This function must not have a parameter list. Note that the result of CLOCK includes the amount of central processor time allocated to all previously executed images.

7.10.3 DATE (str) and TIME (str)

The predeclared procedures DATE and TIME assign the current date and time to a string variable. Each procedure requires a parameter str of type PACKED ARRAY[1..11] OF CHAR.

For example:

```
VAR
    Todays_Date, Current_Time : PACKED ARRAY[1..11] OF CHAR;
    .
    .
    .
DATE (Todays_Date);
TIME (Current_Time);
```

These two calls return results in the following format:

```
 1-Feb-1958
14:20:25.98
```

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

7.10.4 ESTABLISH (function-identifier)

The ESTABLISH procedure establishes a VAX-11 condition handler that processes errors and reports the status of exceptions and conditions. The parameter to ESTABLISH must be the name of a function that has the ASYNCHRONOUS attribute (see Section 10.4). See the *VAX-11 PASCAL User's Guide* for further information.

7.10.5 EXPO (r)

The EXPO function returns the integer-valued exponent of the floating-point representation of the parameter r. When r is of type REAL, SINGLE, or D_floating DOUBLE, the exponent is an integer value from -128 to 127. When r is of type G_floating DOUBLE, the exponent is an integer value between -1024 and 1023. When r is of type QUADRUPLE, the exponent is an integer value between -16,384 and 16,383. The parameter r must be of a real type. (See the *VAX-11 PASCAL User's Guide* for more information about D_floating and G_floating double-precision numbers.)

7.10.6 HALT

The HALT procedure calls the VAX-11 Run-Time Library procedure LIB$STOP with the condition value PAS$_HALT. Without an appropriate condition handler, HALT terminates execution of the program. This procedure must not have a parameter list.

7.10.7 REVERT

The REVERT procedure cancels a condition handler activated by the ESTABLISH procedure. This procedure must not have a parameter list. See the *VAX-11 PASCAL User's Guide* for more information.

Chapter 8
Input and Output

VAX-11 PASCAL includes an extensive set of predeclared routines governing input/output (I/O) processing. These routines enable you to establish files with sequential, relative, or indexed organization and process them by sequential, direct, or keyed access. This chapter describes general I/O processing and the related predeclared routines, and explains the concepts of terminal I/O.

8.1 I/O Processing

The following sections describe in general terms the elements of PASCAL I/O processing: records, files, and access methods. See the *VAX-11 PASCAL User's Guide* for more details.

8.1.1 RMS Records

VAX-11 PASCAL uses the VAX-11 Record Management Services (RMS) subsystem for data storage, retrieval, and modification. Both RMS and PASCAL use the term "file" to define an organized collection of logically related data items. However, PASCAL considers files to consist of file components, while RMS divides files into records. Since RECORD is a predefined structured type in PASCAL, this chapter uses the term "file component" whenever possible. When it is necessary to discuss particular characteristics of RMS records, the term "RMS record" is used.

Generally, a PASCAL file component exactly corresponds to an RMS record. If the file is of a type other than TEXT, an RMS record consists of a single file component. For example, in a file of type INTEGER, each RMS record consists of one integer value. Each I/O statement accesses one file component at a time.

Components of PASCAL text files do not correspond to RMS records. A file of type TEXT has components of type CHAR and is divided into lines. Each line of character components, terminated by an end-of-line marker, constitutes an RMS record.

RMS stores records in one of two formats: fixed length or variable length. Text files are usually, but not necessarily, stored as variable-length RMS records.

8.1.1.1 Fixed-Length RMS Records — In a file composed of fixed-length RMS records, all file components must contain the same number of bytes. You can access fixed-length RMS records with sequential, direct, or keyed access methods. A file with sequential organization that is opened for direct access may contain only fixed-length RMS records to allow the record location to be computed correctly. An indexed file created by VAX-11 PASCAL usually consists solely of fixed-length RMS records.

8.1.1.2 Variable-Length RMS Records — Variable-length RMS records can contain any number of bytes, up to the record length specified when the file was created.Variable-length RMS records are prefixed by a count field whose value indicates the number of bytes in each record. Although any PASCAL file can be created with variable-length RMS records, only text files and files of type VARYING OF CHAR can truly have RMS records of different lengths. All other PASCAL files have components of uniform size.

8.1.2 RMS Files

An RMS file is a collection of logically related components that are arranged in a specific order and treated as a unit. There are three kinds of file arrangement or organization: sequential, relative, and indexed. The organization of a file is determined when the file is created.

Files are normally stored on disk, although sequential files may also be stored on magnetic tape. Other peripheral devices, such as terminals, card readers, and line printers, are treated as sequential files.

8.1.2.1 Sequential Organization — Components of a sequential file are ordered in physical sequence. Each component, except the first, has another component preceding it, and each component, except the last, has another component following it. The physical order in which components appear is identical to the order in which they were written to the file.

8.1.2.2 Relative Organization — Components of a relative file consist of a specified number of fixed-length cells ordered in physical sequence. These cells are numbered from 1 (the first) to n (the last), with each number representing the location of a component relative to the beginning of the file. Each cell either contains a single file component or is empty. You refer to a specific component in the file by its cell number (component number).

8.1.2.3 Indexed Organization — Components of an indexed file are ordered on the basis of certain data fields, called keys, that are contained in each component.

When you design an indexed file, you decide which fields in the file components are to be the keys; the contents of these fields will be used to identify specific components in subsequent operations. The length of a key field and its relative position in the component are identical for all components in the file.

When you create an indexed file, you must define at least one key for the file by using the KEY attribute (see Section 10.11). This mandatory key is called the primary key of the file. By default, the primary key of each component must have a unique value; however, you can change the default to allow duplicate primary keys. You can also define other keys, as many as 254 of them, called alternate keys. An alternate key is a field that is of the same length and in the same position in each component in the file.

8.1.3 Access Methods

The access method is the technique a program uses to retrieve and store file components. VAX-11 PASCAL supports three access methods: sequential, direct, and keyed.

The access method is specified as part of the OPEN procedure, which opens a file. A file's access method cannot be changed unless the file is first closed with the CLOSE procedure and then opened again with a different access method specification.

A file may always be processed sequentially, even when the specified access method is direct or keyed. If the access method is not specified, VAX-11 PASCAL defaults to the sequential method.

Table 8-1 shows the valid access methods for each kind of file organization.

Table 8-1: Access Methods for File Organizations

File Organization	Access Method		
	Sequential	**Direct**	**Keyed**
Sequential	Yes	Yes[1]	No
Relative	Yes	Yes	No
Indexed	Yes	No	Yes

1. Components must be fixed-length RMS records.

8.1.3.1 Sequential Access — Sequential access means that components are processed in sequence. For a sequential file, the sequence is the physical sequence of the components. For a relative file, the sequence is the cell number sequence. For an indexed file, the sequence is the ascending order of primary key values. If two components in an indexed file have the same key value, the sequence is the order of their insertion in the file.

8.1.3.2 Direct Access — Direct access means that the components are processed in an order specified by FIND and LOCATE procedures (see Sections 8.8.2 and 8.8.3). FIND positions a direct-access file to accept input; LOCATE positions the file to write output. A file with sequential organization must have fixed-length RMS records in order to be accessed by the direct method.

8.1.3.3 Keyed Access — Keyed access means that the components are processed in an order determined by the value of a key field. You use the FINDK procedure to indicate the key value of the component you wish to process. FINDK positions the file to the component that corresponds to the key value you specify as a parameter. (See Section 8.9.1 for more information.)

8.2 I/O Procedures

VAX-11 PASCAL provides predeclared procedures and functions to perform input and output operations on file variables. These routines, which may operate differently depending on a file's organization and access method, are arranged in the following categories in this chapter:

General Procedures

- OPEN—opens a VAX/VMS file with specified characteristics
- CLOSE—closes a file

Sequential Access Input Procedures

- GET—reads a file component into the file buffer variable
- READ—reads a file component into a specified variable
- RESET—prepares a file for input

Sequential Access Output Procedures

- PUT—writes the file buffer variable to the specified file
- REWRITE—truncates a file to length zero and prepares it for output
- WRITE—writes specified values to a file

Miscellaneous Routines

- EOF—indicates the end of an input file
- TRUNCATE—truncates records from a file
- UFB—indicates whether the file buffer is undefined
- UNLOCK—unlocks the current component in the file

Text File Manipulation

- EOLN—indicates the end of an input line
- LINELIMIT—terminates program execution after a specified number of lines have been written to a text file
- PAGE—advances output to the next page of a text file
- READLN—reads a line from a text file
- WRITE—allows you to specify field widths to format the values being written to a text file
- WRITELN—writes a line to a text file

Direct Access Procedures

- DELETE—deletes the current component from a file
- FIND—performs direct access to a file for input operations
- LOCATE—performs direct access to a file for output operations
- UPDATE—writes the contents of the file buffer back into the current component

Keyed Access Procedures

- FINDK—accesses a component of an indexed file
- RESETK—readies an indexed file for reading

The I/O procedures (but not the I/O functions) can accept an additional parameter that specifies the action to be taken should the procedure fail to execute successfully. This optional parameter is called ERROR and can accept two values, CONTINUE and MESSAGE. If you specify ERROR:= CONTINUE, the program continues to execute regardless of any error conditions encountered during execution of the procedure. If you specify ERROR:= MESSAGE, an appropriate error message will be printed and program execution will cease if an error occurs. By default, an error message is printed and program execution is terminated after the first error in an I/O operation is encountered.

ERROR must be the last parameter in a procedure's parameter list. You must use nonpositional syntax to call the procedure. You cannot use the ERROR parameter with the I/O functions EOF, UFB, and EOLN, nor with any reference to the file buffer. For further information, consult the *VAX-11 PASCAL User's Guide*.

At any time during the execution of a process, a file variable is considered to be in one of three modes: Inspection, Generation, or Undefined. When a file is reading input, it is in Inspection mode. When output is being written to a file, the file is in Generation mode. A file in an undefined state of processing is in Undefined mode. The mode often determines the valid operations for the file. Table 8–2 shows the mode required before execution of each I/O routine and the mode in which the file is left after each routine has executed.

Table 8-2: File Mode During I/O Processing

I/O Routine	Mode Before Execution	Mode After Execution
OPEN	Undefined	Undefined
CLOSE	Any	Undefined
GET	Inspection	Inspection
READ	Inspection	Inspection
RESET	Any	Inspection
PUT	Generation	Generation
REWRITE	Any	Generation
WRITE	Generation, unless keyed access, which may be any mode	Generation
EOF	Inspection or Generation	No change
STATUS	Any	No change, unless error
TRUNCATE	Inspection	Generation
UFB	Any	No change
UNLOCK	Inspection	Inspection
EOLN	Inspection	Inspection
LINELIMIT	Any	No change
PAGE	Generation	No change
READLN	Inspection	Inspection
WRITELN	Generation	Generation
DELETE	Inspection	Inspection
FIND	Any	Inspection if successful; Undefined if unsuccessful
LOCATE	Any	Generation
UPDATE	Inspection	Inspection
FINDK	Any	Inspection if successful; Undefined in unsuccessful
RESETK	Any	Inspection

8.3 General Procedures

This section describes the following general procedures:

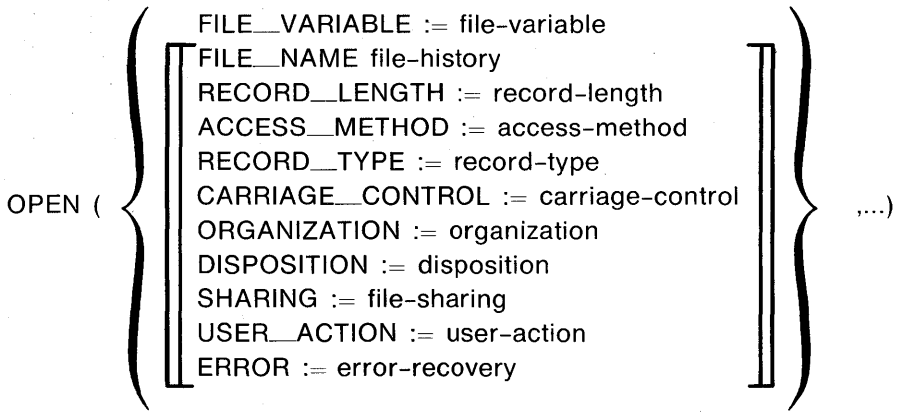
- OPEN
- CLOSE

8.3.1 OPEN Procedure

The OPEN procedure opens a file, defines the file access method, and allows you to specify file parameters. The term "record" in the parameter names of the OPEN procedure indicates an RMS record.

Syntax

1. OPEN (file-variable
   [[file-name]],
   [[history]],
   [[record-length]],
   [[access-method]],
   [[record-type]],
   [[carriage-control]],
   [[organization]],
   [[disposition]],
   [[file-sharing]],
   [[user-action]],
   [[ERROR := error-recovery]])



file-variable

The name of the file variable associated with the file to be opened.

file-name

Information about the file for the operating system.

The file variable and file name designate the file to be opened. Except for the file variable, all parameters are optional. The remaining parameters are summarized in Table 8–3 and discussed in detail in the following sections.

If the parameter names (such as RECORD_TYPE) are not used, as in syntax 1, the parameters must be listed in the specified order. If parameter names are used, as in syntax 2, the parameters can be specified in any order. You can mix the use of positional and nonpositional parameters, but once a nonpositional parameter name has been used, all the following parameter values must be nonpositional.

Table 8–3: Summary of OPEN Procedure Parameters

Parameter	Parameter Values	Default
History	OLD, NEW, READONLY, UNKNOWN	NEW (OLD, if an external file is opened using RESET)
Record-length	Any positive integer value	133 bytes for text files; for other files, parameter is ignored
Access-method	DIRECT, KEYED, or SEQUENTIAL	SEQUENTIAL
Record-type	FIXED or VARIABLE	VARIABLE for new text files and FILE OF VARYING; FIXED for other new files; for old files, record type established at file creation
Carriage-control	LIST, CARRIAGE, FORTRAN, NOCARRIAGE, NONE	LIST for text files and FILE OF VARYING; NOCARRIAGE for all other files. Old files use their existing carriage-control parameter
Organization	SEQUENTIAL, RELATIVE, INDEXED	SEQUENTIAL for new files; previous organization for existing files
Disposition	SAVE, DELETE, PRINT, PRINT_DELETE, SUBMIT, SUBMIT_DELETE	SAVE for named files; DELETE for files without a file-name parameter
Sharing	READONLY, READWRITE, NONE	READONLY if file history is READONLY; NONE for all other files
User-action	Function-identifier	None
Error-recovery	CONTINUE, MESSAGE	MESSAGE (see Section 8.2)

Before the OPEN procedure is called, the file is in Undefined mode; its mode does not change after OPEN has executed.

You cannot use OPEN on a file variable that is already open.

If INPUT and OUTPUT are used, they are implicitly opened when the program begins execution, unless you explicitly open them with OPEN procedures as the first executable statements of the program. INPUT is opened with a history of READONLY unless you specify otherwise.

Because the RESET and REWRITE procedures implicitly open files, you need not always use the OPEN procedure. RESET and REWRITE impose the defaults shown in Tables 8–3 and 8–4. For the file history parameter, RESET uses a default of OLD, and REWRITE uses a default of NEW.

You must use the OPEN procedure to do the following:

- Create a text file with fixed-length RMS records
- Create a file with RELATIVE or INDEXED organization
- Open a file for DIRECT or KEYED access
- Specify a line length other than 133 for a line in a text file

8.3.1.1 File Name — The file name indicates the system name of a file that is represented by a PASCAL file variable in an OPEN procedure. For the file name, you specify a character-string expression (compile-time or run-time) that contains a VAX/VMS file specification or a logical name. (Apostrophes are required to delimit a character-string constant or a logical name used as the file name. See the *VAX-11 PASCAL User's Guide* for more information about logical names.)

If you omit the file name and do not declare the file variable as an external file, the newly created file has no name. If you omit the file name of an external file, the default values shown in Table 8–4 are used.

Table 8–4: 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 its logical name translation
File type	DAT
Version number (history)	OLD: highest current number NEW: highest current number +1

8.3.1.2 History—NEW, OLD, READONLY, or UNKNOWN — The history parameter 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 characteristics. NEW is the default value except when the file has been opened with the RESET procedure.

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 the default value for files opened with the RESET procedure.

A file history of READONLY indicates that an existing file is being opened only for reading. An error occurs if you attempt to write to a file that has been opened with a READONLY file history.

A file history of UNKNOWN indicates that an old file should be opened; if no old file exists, a new file is created with the specified characteristics.

8.3.1.3 Record Length — The value of the record-length parameter is a positive integer that specifies the maximum size in bytes for a line in a text file or a file of type FILE OF VARYING. The default value is 133 bytes. For files of other types, you should not specify a record length.

By default, a file of type TEXT or VARYING OF CHAR has variable-length RMS records. The record length specified for such a file determines the length of the longest line in the file. Each line can contain any number of characters up to the record length specified. If you create a file of type TEXT or VARYING OF CHAR with fixed-length RMS records, the record length determines the exact length of each line in the file. Each line must contain the number of characters specified by the record length.

If you do not specify a record length for an existing file, the length specified at the file's creation is assumed.

8.3.1.4 Access Method—SEQUENTIAL, DIRECT, or KEYED — The access-method parameter specifies the method by which file components are to be accessed. With the SEQUENTIAL method, you can access files with fixed- or variable-length RMS records. The default access method is SEQUENTIAL.

The DIRECT method allows you to use the FIND and LOCATE procedures to gain random access to sequential or relative files with fixed-length RMS records. You cannot use the DIRECT method to access a sequential file that has variable-length records.

With the KEYED method, you can access indexed files using the FINDK procedure to locate a specific component. You cannot open text files for KEYED access.

8.3.1.5 Record Type—FIXED or VARIABLE — The record-type parameter specifies the structure of the RMS records in the file. A value of FIXED indicates that all file components have the same length. A value of VARIABLE indicates that the length of the file components can vary.

VARIABLE is the default record type for a new file of type TEXT or VARYING OF CHAR; other new files use FIXED as the default. For an existing file, the default is the record type associated with the file at its creation.

8.3.1.6 Carriage Control—LIST, CARRIAGE, FORTRAN, NOCARRIAGE, or NONE — The carriage-control parameter specifies the carriage-control format for the file. A value of LIST indicates single spacing between components. LIST is the default option for all text files, including the predeclared file OUTPUT and files of type VARYING OF CHAR.

The CARRIAGE or FORTRAN option indicates that the first character of every output line is a carriage-control character.

The NOCARRIAGE or NONE option specifies that the file has no carriage control. NONE is the default option, except for text files and files of type VARYING OF CHAR.

The effects of the carriage-control options are summarized in Table 8–5 in Section 8.7.5.

8.3.1.7 Organization—SEQUENTIAL, RELATIVE, or INDEXED — The organization parameter specifies the physical organization of a newly created RMS file; it does not determine the manner in which the file is to be accessed. (See Table 8–1 for the valid access methods for each file organization.)

The organization of an existing file must agree with the organization specified when the file is opened. The default value for new files is SEQUENTIAL.

8.3.1.8 Disposition—SAVE, DELETE, PRINT, PRINT_DELETE, SUBMIT, or SUBMIT_DELETE — The disposition parameter describes what is to be done with the file when it is closed. If you specify SAVE, the file is retained. SAVE is the default value for external files.

If you specify DELETE, the file is deleted. If you specify PRINT, the file is submitted to the system line printer spooler and is not deleted. The file is deleted after being printed if you specify PRINT_DELETE.

If you specify SUBMIT, the file is submitted to the batch job queue and is not deleted. The file is deleted after being processed if you specify SUBMIT_DELETE.

An unnamed file is automatically deleted when it is closed and cannot be saved. The only disposition you may specify for an unnamed file is DELETE.

8.3.1.9 Sharing—READONLY, READWRITE, or NONE — The sharing parameter indicates whether other programs can access the file while it is open. A value of READONLY indicates that other programs can read the file while it is open but cannot write to it. READONLY is the default value for files that have a history of READONLY.

A value of READWRITE indicates that other programs can read and write to the file while it is open.

A value of NONE denies other programs all access to the file while it is open. NONE is the default value for files with histories of NEW, OLD, and UNKNOWN.

If you specify SHARING := READWRITE for an existing file with sequential organization, you must explicitly specify ORGANIZATION := SEQUENTIAL in the same OPEN procedure.

8.3.1.10 User Action — The user-action parameter causes the Run-Time Library to call a user-written function to open the file, instead of calling RMS to open the file according to its usual defaults. The user-action parameter allows access to VAX-11 RMS facilities not directly available to a VAX-11 PASCAL program.

A user-action function is expected to perform the RMS tasks that would have been invoked automatically, but it may also perform additional tasks. The required tasks are $OPEN and $CONNECT for existing files, and $CREATE and $CONNECT for new files. The function should return a value indicating whether the file was successfully opened. More extensive information on the user-action parameter is supplied in the *VAX-11 PASCAL User's Guide*.

8.3.1.11 Examples

1.
```
PROGRAM Main (Userguide);

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 the default device and directory on the local computer. If Userguide had not been specified as an external file in the program header, the OPEN procedure would have created an internal file. By default, the file is created with a record length of 133 bytes and RMS 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 parameters for this OPEN procedure has been changed by the use of nonpositional parameter names.

3.
```
OPEN (Solar,
      'Energy',
      HISTORY := NEW,
      RECORD_TYPE := FIXED);
```

This procedure creates a file with the VAX/VMS specification designated by the logical name Energy. The file is created with fixed-length RMS records.

```
4.  OPEN (Journal_Accounts,
          'JOURNAL.DAT',
          HISTORY := UNKNOWN,
          ACCESS_METHOD := KEYED,
          ORGANIZATION := INDEXED);
```

If the file JOURNAL.DAT already exists, this procedure will open it; otherwise, a new file named JOURNAL.DAT will be created with the specified characteristics. If the file does exist, it must have the same characteristics as those in the parameter list of the OPEN procedure. The file is opened with indexed organization for keyed access.

```
5.  OPEN (CheckingBalance,
          ORGANIZATION := RELATIVE,
          ACCESS_METHOD := DIRECT,
          USER_ACTION := Open_Checking);
```

This procedure opens the file CheckingBalance by calling the user-action function Open_Checking. The Open_Checking function should perform the RMS tasks $CREATE and $CONNECT, in addition to any other operations. The function returns a value indicating whether the file was successfully opened with relative organization for direct access.

8.3.2 CLOSE Procedure

The CLOSE procedure closes an open file.

Syntax

1. CLOSE (file-variable
   [[disposition]],
   [[user-action]],
   [[ERROR := error-recovery]])

2. CLOSE ({ FILE_VARIABLE := file-variable
   [[DISPOSITION := disposition
   [[USER_ACTION := user-action
   [[ERROR := error-recovery]]]]]] } ,...)

file-variable

The name of the file variable associated with the file to be closed.

Except for the file variable, all parameters to the CLOSE procedure are optional. If the nonpositional parameter names are not used, as in syntax 1, the parameters must be in the order specified. If nonpositional parameter names are used, as in syntax 2, the parameters can be specified in any order.

The file may be in any mode before the CLOSE procedure is called. Execution of CLOSE sets the mode to Undefined.

Execution of the CLOSE procedure causes the system to close the file and, if the file is internal, to delete it. Each file is automatically closed when control passes 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. If you attempt to close a file that was never opened, an error occurs.

8.3.2.1 Disposition—SAVE, DELETE, PRINT, PRINT_DELETE, SUBMIT, or SUBMIT_DELETE — The disposition parameter describes what is to be done with the file when it is closed. The parameter values and the defaults are the same as those for the disposition parameter in the OPEN procedure (refer to Table 8–4).

If a disposition value was specified in the OPEN procedure, an identical disposition value is usually specified in the CLOSE procedure. If the two values are different, the value in the CLOSE procedure takes precedence.

8.3.2.2 User Action — The user-action parameter causes the Run-Time Library to call a user-written function to close the file instead of closing the file according to its usual defaults. The user-action parameter allows access to VAX–11 RMS facilities not explicitly available to a PASCAL program.

A user-action function is expected to perform the RMS $CLOSE task that would have been invoked automatically, but it may perform additional tasks. The function should return a value indicating whether the file was successfully closed. More extensive information on the user-action parameter is supplied in the *VAX–11 PASCAL User's Guide*.

8.3.2.3 Examples

1.
```
CLOSE (Albums);
```

This procedure closes the file Albums and deletes it if it is an internal file.

2.
```
CLOSE (Products,
       DISPOSITION := PRINT_DELETE);
```

This procedure closes the files Products, submits it to the line printer, and deletes it after the hard copy is produced. The file must not have been opened with a file history of READONLY.

3.
```
CLOSE (ShoeInventory,
       USER_ACTION := Close_File);
```

This procedure calls the user-action function Close_File, which must perform the RMS task $CLOSE in addition to any other operations. The function must return a value to indicate whether the file ShoeInventory was successfully closed.

8.4 Sequential Access Input Procedures

This section describes input procedures that apply primarily to files opened for sequential access; however, these procedures can also be used on files opened for direct and keyed access.

The sequential access input procedures are:

- GET
- READ
- RESET

8.4.1 GET Procedure

The GET procedure advances the file position and reads the next component of the file into the file buffer variable. If the file has relative or indexed organization, the component is also locked.

Syntax

GET (file-variable [[, ERROR := error-recovery]])

file-variable

The name of the file variable associated with the input file.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the GET procedure is executing (see Section 8.2).

Before the GET procedure is used for the first time to read one or more file components, the file must be in Inspection mode and prepared for reading input. Depending on the access method specified when the file was opened, you can prepare the file for input in the following ways:

- If the file is open for sequential access, call the RESET procedure. RESET sets the mode to Inspection, advances the file position to the first component, and assigns the component's value to the file buffer variable.
- If the file is open for direct access, call either the RESET or FIND procedure, either of which positions the file.
- If the file is open for keyed access, call the FINDK, RESET, or RESETK procedure to position the file.

As a result of the GET procedure, the file remains in Inspection mode, and the file position advances to the next component. This component is locked and EOF and UFB are set to FALSE. Unless the end-of-file marker is encountered, the file buffer variable takes on the value of that component. If no component is found, EOF and UFB are set to TRUE. The following example shows the use of GET:

```
RESET (Books);
Newrec := Books^;
GET (Books);
```

After execution of the RESET procedure, the value of the file buffer variable Books^ is equal to the value of 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^, advancing the file position to the second component. Another GET procedure would advance the file position to the third component. This sequence of events is illustrated in Figure 8–1.

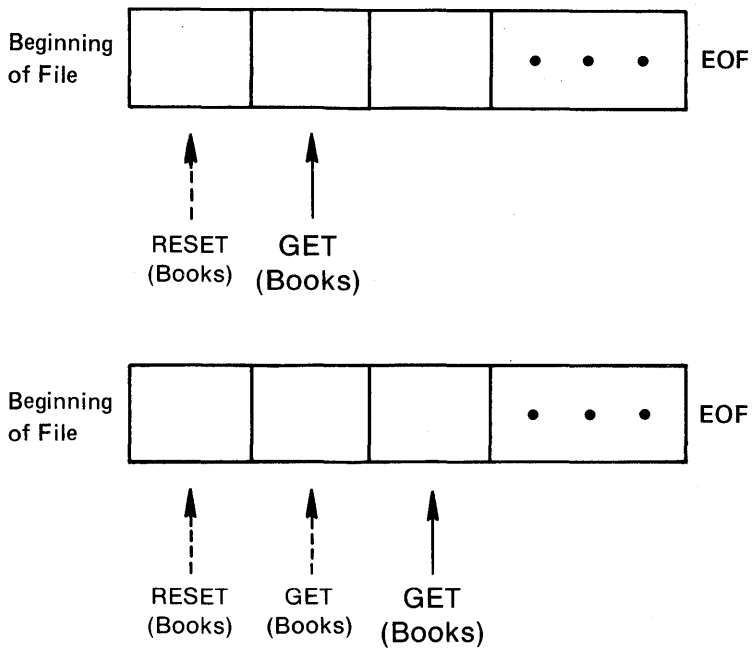


Figure 8–1: File Position After GET

By using the GET procedure repeatedly, you can read sequentially through a file.

When called for a file with relative organization, GET skips any nonexistent components to find the next component. A successful GET operation locks the component and sets EOF and UFB to FALSE. If a component is not found, EOF and UFB become TRUE.

When you reach the end of the file, EOF automatically becomes TRUE and the file buffer variable becomes undefined (UFB is TRUE). If GET is used when EOF is TRUE, a run-time error occurs and program execution is aborted.

Example

```
GET (Phones);
```

This example reads the next component of the file Phones into the file buffer variable Phones^. Prior to executing GET, the value of EOF (Phones) must be FALSE; if it is TRUE, an error occurs.

8.4.2 READ Procedure

The READ procedure reads one or more file components into a variable.

Syntax

READ ([[file-variable,]] {variable-identifier},... [[, ERROR := error-recovery]])

file-variable

The name of the file variable associated with the input file. If you omit the name of the file, the default is INPUT.

variable-identifier

The name of the variable into which a file component will be read; multiple identifiers must be separated with commas.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the READ procedure is executing (see Section 8.2).

The file must be in Inspection mode before READ is called. The file remains in Inspection mode after execution of a READ procedure.

By definition, the READ procedure for a nontext file performs an assignment statement, a GET procedure, and an UNLOCK procedure for each variable. Thus, the procedure call

```
READ (file-variable, variable-identifier);
```

is similar to

```
variable-identifier := file-variable^;
GET (file-variable);
UNLOCK (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 read is assigned to the second variable, and so on. The values and the variables must be of assignment-compatible types. Reading stops if an error occurs.

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, GET, and UNLOCK process until it has read a sequence of characters that represent a legal value for the next variable in the parameter list. The procedure 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. Unless you are reading into a character or string variable, a call to READ at this point skips over the end-of-line marker and positions the file at the beginning of the next line. If you are reading into a variable of type CHAR when EOLN is TRUE, the space is read and assigned to the variable, and the file position advances. If you are reading into a string variable when EOLN becomes TRUE, the file position does not change. In the latter case, you should do a READLN to advance the file position past the end-of-line marker.

Values from a text file can be read into variables of integer, real, character, string, and enumerated types. Text file values to be read into integer, real, and enumerated-type variables can be preceded in the file by any number of spaces, tabs, and end-of-line markers. Values to be read into character variables, however, must not be separated because they are read and assigned character by character. If an invalid character is encountered during the reading of a text file item, the value being formed is terminated.

When reading constant identifiers of an enumerated type from a text file, the PASCAL run-time system reads all characters in the identifier but recognizes only the first 31 characters. You need input only enough characters to make the identifier unique among the other constant identifiers of its type. Text input data for enumerated types may consist of both lower- and uppercase characters.

Boolean input data in text files obey the same rules as other enumerated types. For example, all of the following character combinations that could appear in a text file are equivalent: TRUE, True, T, t, tr.

You can use the READ procedure to read a sequence of characters from a text file into a variable of type PACKED ARRAY OF CHAR. Successive characters from the file are assigned to components of the array, in order, until each component has been assigned a value. If any characters remain on the line after the array is full, the next READ procedure begins with the next character on that line. If the end of the line is encountered before the array is full, spaces are assigned to the remaining components.

You can also read text file characters into a variable of type VARYING OF CHAR. Characters are assigned to a VARYING string in a manner similar to that in which they are assigned to a packed array. However, if the end-of-line marker is encountered before the VARYING string has been filled to its maximum length, the VARYING string value is not padded with spaces. Instead, its current length is set equal to the number of characters that have been read into it. If you call the READ procedure with a parameter of type VARYING OF CHAR and EOLN is TRUE, no characters are read into the VARYING string; its current length is set to zero.

Every nonempty 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 a 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, and that the following values have been entered at the terminal:

```
98.6 11 75
```

   The variable Temp is assigned the value 98.6, Age is assigned the value 11.0, and Weight is assigned the value 75.0. 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 character strings:

```
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, after encountering the end-of-line marker, fills the array Veep with spaces. The file position will not advance to the beginning of the next line until a READLN is performed.

3.
```
TYPE
   Color = (Red, Fire_Engine_Green, Blue, Black);

VAR
   Light : Color;
   .
   .
   .
READ (Light);
```

In this example, if the letter R is read, the variable Color is assigned the value Red. However, if the letters Redx are read, an error occurs. If the letters Bl are read, an error also occurs since Bl is not unique. However, the letters Blu are unique and would be interpreted as the constant identifier Blue.

8.4.3 RESET Procedure

The RESET procedure readies a file for reading.

Syntax

RESET (file-variable [, ERROR := error-recovery])

file-variable

The name of the file variable associated with the input file.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the RESET procedure is executing (see Section 8.2).

The file may be in any mode before RESET is called to set the mode to Inspection.

If the file is an external file and is not already open, RESET opens it using the defaults listed in Tables 8–3 and 8–4. You cannot use RESET to create a file.

After execution of RESET, the file is positioned at the first component, and the file buffer variable contains the value of this component. If the file is not empty, EOF and UFB are FALSE and the first component is locked to prevent access by other processes. If the file is empty, EOF and UFB are TRUE. If the file does not exist, RESET does not create it, but returns an error at run time.

You should call RESET before reading any file with sequential organization except the predeclared file INPUT. The RESET procedure removes the end-of-file marker from any file connected to a terminal device (including INPUT), thus allowing reading from the file to continue. If you call RESET for the predeclared file OUTPUT, an error occurs.

A call to RESET on a relative file opened for direct access positions the file at its first existing component.

A call to RESET on an indexed file opened for keyed access positions the file at the first component relative to the primary key.

Examples

1.
```
OPEN (Phones,
      'Phones.Dat',
      ACCESS_METHOD := Direct);
RESET (Phones);
```

These statements open the file variable Phones for direct access. 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 procedure call prepares it for reading and assigns the value of the first file component to Weights^. If the file is not open, RESET causes the system to perform an OPEN by default. If Weights is an external file, its file history will be OLD. Otherwise, an error occurs.

8.5 Sequential Access Output Procedures

This section describes output procedures that apply primarily to files opened for sequential access; however, these procedures can also be used on direct- and keyed-access files.

The following sequential output procedures are described:

- PUT
- REWRITE
- WRITE

8.5.1 PUT Procedure

The PUT procedure adds a new component to a file.

Syntax

PUT (file-variable [, ERROR := error-recovery])

file-variable

The name of the file variable associated with the output file.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the PUT procedure is executing (see Section 8.2).

Before executing the first PUT procedure on a sequential-access file, you must execute a REWRITE or TRUNCATE procedure to set the file to Generation mode. Both REWRITE and TRUNCATE set EOF to TRUE, thus preparing the file for output. (Note that TRUNCATE is legal only on files with sequential organization; see Section 8.6.3.) If the file has indexed organization, the components to be written must be ordered by primary key.

Before executing the first PUT on a file opened for direct access, you must execute a REWRITE or LOCATE procedure to position the file.

The PUT procedure writes the value of the file buffer variable at the end of the specified sequential- or direct-access file. After execution of the PUT procedure, the value of the file buffer variable becomes undefined (UFB is TRUE). EOF remains TRUE and the file remains in Generation mode.

You may call the PUT procedure for a keyed-access file, regardless of the file's mode (Inspection, Generation, or Undefined). PUT causes the file buffer variable to be written to the file at the position indicated by the key. If the component has more than one key, the file buffer variable is inserted in each index at the appropriate location. After execution of PUT, a keyed-access file is in Generation mode.

Example

```
PROGRAM Bookfile (INPUT, OUTPUT, Books);

TYPE
   String = PACKED ARRAY[1..40] OF CHAR;
   Bookrec = RECORD
             Author : String;
             Title : String;
             END;

VAR
   Newbook : Bookrec;
   Books : FILE OF Bookrec;
   N : INTEGER;

BEGIN
REWRITE (Books);
FOR N := 1 TO 10 DO
   BEGIN
   WITH Newbook DO
      BEGIN
      WRITE ('Title:');
      READLN (Title);
      WRITE ('Author:');
      READLN (Author);
      END;
   Books^ := Newbook;
   PUT (Books);
   END;
CLOSE (Books);
END.
```

This program writes the first 10 records read from the terminal into the file Books. The records are typed at the terminal and assigned 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 input record.

8.5.2 REWRITE Procedure

The REWRITE procedure readies a file for output.

Syntax

REWRITE (file-variable [, ERROR := error-recovery])

file-variable

The name of the file variable associated with the output file.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the REWRITE procedure is executing (see Section 8.2).

The file can be in any mode before REWRITE is called to set the mode to Generation. If the file variable has not been opened, REWRITE creates and opens it using the defaults listed in Tables 8–3 and 8–4.

The REWRITE procedure truncates a file to length zero and sets EOF and UFB to TRUE. You can then write new components into the file with the PUT, WRITE, and WRITELN procedures (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 with sequential organization, you must either use the TRUNCATE procedure or copy the contents to another file, specifying new values for the components you need to update.

REWRITE, when applied to a file with relative or indexed organization, deletes the contents of the file and sets the file position to the beginning of an empty file.

Examples

1.
```
REWRITE (Storms);
```

If the file variable Storms is already open, this REWRITE procedure prepares the file for writing, clears it 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 8.3.1).

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 to the beginning of the file. After execution of this procedure, EOF (Ratings) is TRUE.

8.5.3 WRITE Procedure

The WRITE procedure assigns data to an output file.

Syntax

WRITE ([[file-variable,]]{expression},... [[, ERROR := error-recovery]])

file-variable

The name of the file variable associated with the output file. If you omit the name of the file, the default is OUTPUT.

expression

A compile-time or run-time expression whose value is to be written; multiple output values must be separated with commas. An output value must have the same type as the file components; however, values written to a text file can also be expressions of any ordinal, real, or string type.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the WRITE procedure is executing (see Section 8.2).

The file (unless it is a keyed-access file) must be in Generation mode before WRITE is called; it remains in that mode after WRITE has executed.

By definition, a WRITE to a nontext file performs an assignment to the file buffer variable and a PUT for each output value. For nontext files, the types of the output values must be assignment compatible with the component type of the file. Thus, the procedure call

```
WRITE (file-variable, expression);
```

is similar to

```
file-variable^ := expression;
PUT (file-variable);
```

For text files, the WRITE procedure converts the value of each expression to a sequence of characters. It repeats the assignment and PUT process until all the values have been written to the file. See Section 8.7.8 for information on using WRITE with text files.

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 value of the string variable Pres.

2.
```
VAR
    Rain_Amts : 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 rainfall. The WRITE procedure writes the values of the variables Avg_Rain and Min_Rain into the file, and follows them with the real constant 0.312 and the value of the variable Max_Rain.

8.6 Miscellaneous Routines

The miscellaneous routines described in this section are generally used when dealing with sequential access files. In some cases, as indicated, the routines can also be used on direct or keyed access files.

- EOF (also legal on files opened for direct or keyed access)
- STATUS (also legal on files opened for direct or keyed access)
- TRUNCATE
- UFB (also legal on files opened for direct access)
- UNLOCK (also legal on files opened for direct or keyed access)

8.6.1 EOF Function

The EOF (end-of-file) function indicates whether the file pointer is positioned after the last component in a file.

Syntax

EOF [[(file-variable)]]

file-variable

The name of the file variable associated with the input file. If you omit the name of the file, the default is INPUT.

The file may be in either Inspection or Generation mode before EOF is called; however, end-of-file must be well defined. The input operations GET, RESET, FIND, and FINDK are guaranteed to leave end-of-file well defined. The file mode does not change after EOF has executed.

The Boolean function EOF returns TRUE when the file pointer is positioned after the last component in the file. The EOF function returns FALSE up to and including the time when the last component of the input file is read into the file buffer. You must attempt to get another file component after the last to determine whether the file is positioned at end-of-file.

When EOF is tested for a file with relative organization opened for direct access, the result is TRUE if the file is in Inspection mode and the last GET or RESET operation positioned the file beyond the last existing component. If the file is in Generation or Undefined mode, the result of EOF is undefined.

When EOF is tested for a file with indexed organization opened for keyed access, the result is TRUE if the file is in Inspection mode and the last FINDK, GET, RESET, or RESETK operation positioned the file beyond the last component with the current key number. Successful attempts at FINDK, GET, RESET, and RESETK cause EOF to be FALSE. If the file is not in Inspection mode, EOF is undefined.

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 while the EOF function returns FALSE.

2.
```
WHILE NOT EOF (MasterFile) DO
   BEGIN
   READLN (MasterFile, Customer);
   IF Customer.New <> Yes
   THEN
      Old := Old + 1
   ELSE
      New := New + 1;
   END;
```

   This example counts the numbers of old and new customers in a master file. The loop is performed while EOF is FALSE.

8.6.2 STATUS Function

The STATUS function indicates the status of a file following the last operation performed on it.

Syntax

STATUS (file-variable)

file-variable

The name of the file variable associated with the file to be tested.

The file may be in any mode before STATUS is called; unless an error occurs, STATUS does not change the file mode upon execution.

The STATUS function returns one of the following integer codes that indicate the previous operation's effect on the file: 0 indicates a successful operation; -1 indicates that the previous operation encountered an end-of-file; a positive integer value indicates that the previous operation resulted in an error. The specific error condition codes returned by the STATUS function are listed in the *VAX-11 PASCAL User's Guide*.

STATUS never signals an error condition using the VAX-11 Condition Handling Facility; rather, it reports an error status in its return value.

A test by the STATUS function on a text file causes delayed device access to occur, thus filling the file buffer with the next file component (see Section 8.10). Therefore, EOF, EOLN, UFB, and STATUS never return an error code following a successful STATUS function.

Example

```
RESET (File1, ERROR := CONTINUE);
IF STATUS (File1) > 0
THEN
   WRITELN ('Cannot access first record')
ELSE
   IF STATUS (File1) < 0
   THEN
      WRITELN ('File is empty')
   ELSE
      READ (File1);
```

This example resets a file and prepares it for reading. Following the RESET, the file status is tested first for an error condition and then for an end-of-file. If the RESET procedure encounters either of these conditions, an appropriate error message is printed. If the STATUS function indicates that the RESET was successful, the first record is read from the file.

8.6.3 TRUNCATE Procedure

The TRUNCATE procedure indicates that the current file component and all components following it are to be deleted.

Syntax

TRUNCATE (file-variable [, ERROR := error-recovery])

file-variable

The name of the file variable associated with the file to be truncated.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the TRUNCATE procedure is executing (see Section 8.2).

The file must be in Inspection mode before TRUNCATE is called. After the procedure's execution, the mode is set to Generation so that output can be written to the file.

The current component is the one at which the file buffer is positioned. After the appropriate components have been deleted, the file remains positioned at the new end-of-file, but the file buffer itself is undefined. Thus, EOF and UFB are both set to TRUE.

TRUNCATE can be used only on a file that has sequential organization.

Example

```
TRUNCATE (MasterFile);
```

This procedure deletes components from the sequential file MasterFile, beginning with the current component and continuing until EOF is TRUE. When the operation is complete, EOF (MasterFile) and UFB (MasterFile) are TRUE and new data may be written at the end of MasterFile.

8.6.4 UFB Function

The UFB (undefined file buffer) function returns a Boolean value to indicate whether the last file operation gave the file buffer an undefined status.

Syntax

UFB (file-variable)

file-variable

The name of the file variable associated with the file whose buffer is being tested.

The file may be in any mode before UFB is called; execution of UFB does not change the file mode.

UFB tests the effect of the last I/O operation done to the file. UFB returns FALSE if a successful GET, FIND, FINDK, RESET, or RESETK operation has filled the file buffer. GET, FIND, FINDK, RESET, and RESETK procedure calls that do not fill the file buffer set UFB to TRUE. UFB also returns TRUE after DELETE, LOCATE, PUT, REWRITE, TRUNCATE, and UPDATE procedures have left the contents of the file buffer unknown.

Assigning a new value to the file buffer with an assignment statement does not change the value of UFB.

Example

```
FIND (Supplies, December);
IF NOT UFB (Supplies)
THEN
   Inventory := Inventory - Supplies^;
```

If the variable December has a value of 12, the FIND procedure attempts to find the twelfth component of the file Supplies. If the FIND procedure is successful, Supplies^ assumes the value of this component and UFB (Supplies) is FALSE. If, however, the FIND procedure is unable to find the twelfth component of the file, UFB (Supplies) returns TRUE. In this example, the value of Supplies^ is subtracted from the value of Inventory only if the FIND procedure is successful.

8.6.5 UNLOCK Procedure

The UNLOCK procedure releases the current file component for access by other processes.

Syntax

UNLOCK (file-variable [, ERROR := error-recovery])

file-variable

The name of the file variable associated with the file whose component is to be unlocked.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the UNLOCK procedure is executing (see Section 8.2).

The file must be in Inspection mode before UNLOCK is called; it remains in Inspection mode after UNLOCK has executed.

If the component at which the file pointer is positioned has been locked, the UNLOCK procedure releases it.

Although UNLOCK may be used on files with any organization, no unlocking is performed on files with sequential organization. When such a file is opened, it is locked as a whole, rather than by individual components, with the SHARING parameter. However, a call to the UNLOCK procedure for a sequential file does not cause an error.

Example

```
UNLOCK (SalesFile);
```

If SalesFile has direct or indexed organization, the UNLOCK procedure releases the contents of the current component. If SalesFile has sequential organization, the procedure has no effect.

8.7 Text File Manipulation

The following routines apply only to the handling of text files (including INPUT and OUTPUT). The following routines are described:

- EOLN
- LINELIMIT
- PAGE
- READLN
- WRITELN

In addition, the use of the output procedures WRITE, WRITELN, and WRITEV with a field width specification for more readable output is described in Sections 8.7.6 and 8.7.7, and prompting on terminal files is discussed in Section 8.7.8. (The WRITEV procedure, which writes the values of expressions to a VARYING string, is fully described in Section 7.6.9.)

8.7.1 EOLN Function

The EOLN (end-of-line) function tests for the end-of-line marker within a text file.

Syntax

EOLN [[(file-variable)]]

file-variable

The name of a file variable associated with a text file. If you omit the name of the file, the default is INPUT.

The file must be in Inspection mode and EOF must be FALSE before EOLN is called. EOLN leaves the file in Inspection mode.

The Boolean EOLN function returns TRUE when the file pointer is positioned after the last character in a line. When EOLN is TRUE, the file buffer contains a blank character.

The EOLN function returns FALSE when the last component in the line is read into the file buffer. Another character must be read to cause EOLN to return TRUE and to cause the file buffer to be positioned at the end-of-line marker following the last character of the line. If you use the EOLN function on a nontext 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 it calculates the number of characters in that line. The WHILE statement continues to execute until the end-of-line marker is read.

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

This example scans the characters on each line of a text file called MasterFile and checks for characters that are neither digits nor letters. If a nonnumeric or nonalphabetic character is encountered in the file, the counter Err is incremented by one. The loop is executed until the last component in the file is read.

8.7.2 LINELIMIT Procedure

The LINELIMIT procedure terminates execution of the program after a specified number of lines has been written into a text file.

Syntax

LINELIMIT (file-variable, n [, ERROR :=error-recovery])

file-variable

The name of the file variable associated with the text file to which this limit applies.

n

A positive integer expression that indicates the number of lines that can be written to the file before execution terminates.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the LINELIMIT procedure is executing (see Section 8.2).

The file may be in any mode before LINELIMIT is called; the file mode does not change after LINELIMIT has executed.

The VAX-11 PASCAL run-time system determines a default line limit for text files by translating the logical name PAS$LINELIMIT as a string of decimal digits. If this logical name has not been defined, there is no default line limit. You can override the default by calling LINELIMIT with a smaller or larger value for n.

After the number of lines written into the file has reached the line limit, program execution terminates unless the WRITELN procedure that exceeded the line limit includes the ERROR:=CONTINUE parameter.

Example

```
LINELIMIT (Debts, 100);
```

Execution of the program terminates after 100 lines have been written into the text file Debts.

8.7.3 PAGE Procedure

The PAGE procedure skips from the current page to the next page of a text file.

Syntax

PAGE (file-variable [, ERROR := error-recovery])

file-variable

The name of a file variable associated with a text file.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the PAGE procedure is executing (see Section 8.2).

The file must be in Generation mode before the PAGE procedure is called; the mode does not change as a result of the procedure's execution.

Execution of the PAGE procedure requires the system to clear the record buffer, if it contains data, by performing a WRITELN, and then to advance the output to a new page of the specified text file. The next line written to the file begins on the second line of a new page (the first line is always empty). You can use this procedure only on text files. If you specify a file of any other type, an error occurs.

The value of the page eject record that is output to the file depends on the carriage-control format for that file. When CARRIAGE or FORTRAN is enabled, the page eject record is equivalent to the carriage control character '1'. When LIST, NOCARRIAGE, or NONE is enabled, the page eject record is a single form-feed character.

Examples

1. ```
   PAGE (Userguide);
   ```

   This PAGE procedure causes a page eject record to be written in the text file Userguide.

2. ```
   PAGE (OUTPUT);
   ```

   This PAGE procedure writes a page eject record to the terminal (in interactive mode) or in the batch log file (in batch mode).

8.7.4 READLN Procedure

The READLN procedure reads lines of data from a text file.

Syntax

READLN ([file-variable,] {variable-identifier},... [, ERROR := error-recovery])

file-variable

The name of the file variable associated with the text file to be read. If you omit the name of the file, the default is INPUT.

variable-identifier

The name of the variable into which a value will be read; multiple identifiers must be separated with commas. If you do not specify any variable names, READLN skips a line in the specified file.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the READLN procedure is executing (see Section 8.2).

The file must be in Inspection mode before READLN is called; it remains in that mode after the procedure's execution.

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.

When applied to several variables, READLN performs the following sequence:

```
READ (file-variable, {variable-identifier},...);
READLN (file-variable);
```

EOLN is TRUE after a READLN only if the new line is empty.

You can use the READLN procedure to read integers, real numbers, characters, strings, and constants of enumerated types. The values in the file must be separated as for the READ procedure. The rules governing the reading of values from text files are presented with the READ procedure (see Section 8.4.2).

Example

```
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 the space detected as the end-of-line marker to Veep. A third call to the procedure reads 'Richard M. Nixon ' into Pres and 'Spiro T. Agnew ' into Veep. The procedure then skips past the end-of-line marker to the beginning of the next line. Another call to READLN sets EOLN and EOF equal to TRUE.

8.7.5 WRITELN Procedure

The WRITELN procedure writes a line of data to a text file.

Syntax

WRITELN [([file-variable,]{expression},... [, ERROR := error-recovery])]

file-variable

The name of the file variable associated with the text file to be written. If you omit the name of the file, the default is OUTPUT.

expression

A compile-time or run-time expression whose value is to be written; multiple output values must be separated by commas. The expressions can be of any ordinal, real, or string type and are written with a default field width (see Section 8.7.6).

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the WRITELN procedure is executing (see Section 8.2).

The file must be in Generation mode before WRITELN is called; it remains in that mode after WRITELN has executed.

The WRITELN procedure writes the specified values into the text file, inserts an end-of-line marker after the end of the current line, and then positions the file at the beginning of the next line. When applied to several expressions, WRITELN performs the following sequence:

```
WRITE (file-variable, {expression},...);
WRITELN;
```

For example:

```
WRITELN (Userguide, 'This manual describes how to interact');
```

As a result of this procedure, the string is written to the text file Userguide, followed by an end-of-line marker. and skips to the next line.

When you open a text file or a file of type VARYING OF CHAR, you can specify the value CARRIAGE (or FORTRAN) for the carriage-control parameter. If you select CARRIAGE (or FORTRAN) 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 a terminal or a line printer. If output is directed elsewhere, the character is written into the file and will be read back when the file is opened for input.

Table 8–5 summarizes the carriage-control characters and their effects. For purposes of carriage control, any characters other than those listed in the table are ignored.

Table 8–5: Carriage-Control Characters

Character	Meaning
'+'	Overprinting: starts output at the beginning of the current line
' '	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
''(0)	Prompting with overprinting: suppresses line feed at the beginning of the line and carriage return at the end of the line; note that this character is the ASCII NUL character

The carriage-control character must be the first item in an output text line. For example, if the text file Tree has been opened with the CARRIAGE option, you can use the following procedure:

```
WRITELN (Tree, ' ', String1, String2);
```

The first item in the list is a space character. The space indicates that the values of String1 and String2 will be printed on a new line when the file is written to a terminal, line printer, or similar carriage-control device.

If you select CARRIAGE format when opening the predeclared file OUTPUT, you can use the dollar sign ($) character to initiate prompting for input at the terminal. For example:

```
WRITELN ('$How many inches of rain last night?');
```

This procedure 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[I], ' is the grade for this student.');
```

   This WRITELN procedure writes a component of the character array Class to the file OUTPUT.

2.
```
WRITELN;
```

   A call to WRITELN without a file variable or print list ends the printing of the current line on the file OUTPUT, which represents the standard output device (usually the terminal).

3.
```
TYPE
   String = PACKED ARRAY[1..25] OF CHAR;

VAR
   Newhires : TEXT;
   N : INTEGER;
   Newrec : RECORD
            Id : INTEGER;
            Name, Address : String;
            END;
   .
   .
   .
OPEN (Newhires,
     CARRIAGE_CONTROL := CARRIAGE);
REWRITE (Newhires);
WITH Newrec DO
   BEGIN
   WRITELN (Newhires, '1New hire #  ', ID:1, ' is ', Name);
   WRITELN (Newhires, ' ', Name, 'lives at:');
   WRITELN (Newhires, ' ');
   WRITELN (Newhires, ' ', Address);
   END;
```

   In this example, four lines are written to the text file Newhires. The output starts at the top of a new page, as directed by the carriage-control character '1', and appears in the following format:

```
New hire #  73 is Irving Washington
Irving Washington         lives at:

22 Chestnut St, Seattle
```

8.7.6 Output with Specified Field Width

The output values of a WRITE, WRITELN, or WRITEV (see Section 7.6.9) procedure can be compile-time or run-time expressions, with values of any ordinal, real, or string type. Each value is written with a default field width, which specifies the minimum number of characters to be written for the value. Table 8–6 lists the default field widths.

Table 8–6: Default Field Widths

Type of Item Printed	Number of Characters
INTEGER, UNSIGNED	10
CHAR	1
BOOLEAN	6
Enumerated	Size of longest identifier +1 up to 32
REAL	12
DOUBLE	20
QUADRUPLE	40
Character String	Length of string

You can override these defaults for a particular value by specifying a field width in the print list, using the following format:

output:minimum[[:fraction]]

Both minimum and fraction represent integer expressions with positive or zero values. The minimum indicates the minimum number of characters to be written for the value. The fraction, which is permitted only for values of real types, indicates the number of digits to be written to the right of the decimal point. The format of the field width specification is identical for the WRITE, WRITELN, and WRITEV procedures.

By default, real numbers are written in exponential format. Note that regardless of the real number's type, output procedures always prefix the exponent with the letter E. Each real number in exponential format is preceded by a blank or a minus sign, and the value of the rightmost digit is rounded. For example:

```
WRITELN (Shoesize);
```

If the value of Shoesize is 12.5, this procedure produces the following output:

```
 1.25000E+01
```

To write the value in decimal format, you must specify a field width as in this example:

```
WRITELN (Shoesize:5:1);
```

The first integer indicates that a minimum of five characters will be written. The minimum includes the minus sign, if needed, and the decimal point. The second integer specifies one digit to the right of the decimal point. The resulting output is as follows:

```
12.5
```

If the field specified is wider than necessary, the value is written with leading blanks.

If you try to write an integer, unsigned, or real value in a field that is too narrow, the field width is expanded to the minimum necessary to write the value. If you try to write a value of an enumerated type, a Boolean value, or a string value in a field that is too narrow, the value is truncated on the right. The truncated identifier is not checked for uniqueness.

For an expression of an enumerated type, the constant identifier denoting the expression's value is written. 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 written:

```
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 for all 17 characters in the identifier, the identifier is truncated after the field is filled. Note that constants of enumerated types are written in all uppercase characters.

8.7.7 Writing Binary, Hexadecimal, and Octal Values

You can use the predeclared conversion functions BIN, HEX, and OCT in combination with the WRITE, WRITELN, and WRITEV procedures to write binary, hexadecimal, and octal values. These functions and the WRITEV procedure are described in detail in the subsections of Section 7.6.

Syntax

```
WRITE ([file-variable,]
   {BIN (expression[, length[, digits]])},...)

WRITE ([file-variable,]
   {HEX (expression[, length[, digits]])},...)

WRITE ([file-variable,]
   {OCT (expression[, length[, digits]])},...)
```

The BIN, HEX, and OCT functions convert the value of the first expression in the list to its equivalent as a binary, hexadecimal, or octal number. The resulting digits are returned in a VARYING string.

The actual parameter list of the conversion function must contain an expression to be written. Two optional integer parameters specify the length of the resulting string and the number of significant digits to be returned. If you omit these parameters, the bit width of the converted value determines the string length and the number of significant digits. If the converted value is shorter than the specified length, it is padded with spaces on the left. If the converted value is longer, it is truncated on the left.

For every expression whose binary, hexadecimal, or octal value you wish to write, you must call the appropriate conversion function separately with an actual parameter list. You can call more than one BIN, HEX, or OCT function in the same output procedure call. Arbitrary items (including pointers) may be written in binary, hexadecimal, or octal notation to text files.

You can specify field widths with the BIN, HEX, and OCT functions; however, the results are likely not to be what you expect. For example, suppose you want to convert the value of I to its hexadecimal equivalent and you want the converted value to be written in a field three characters wide. You might write the following procedure call:

```
WRITELN (HEX (I):3);
```

However, since the converted value is longer than the field width specification, the value is truncated on the right rather than on the left. Therefore, the output generated by this procedure would be:

```
 00
```

Thus, you should be careful about specifying field widths with BIN, HEX, and OCT when the converted value could exceed the field width given.

Examples

1.
```
WRITE (HEX (Payroll, 10), HEX (Salary, 12));
```

   The values of the variables Payroll and Salary are converted to their hexadecimal equivalents. Payroll is printed with 10 characters and Salary is printed with 12 characters. The output values, preceded by two initial blanks, might look like this:

```
000031F2     000058AB
```

2.
```
WRITELN (OCT (Social_Security, 14), BIN (Survey, 8));
```

   The value of the variable Social_Security is converted to its octal equivalent and printed with 14 characters. Then the value of the variable Survey is converted to its binary equivalent and printed with eight characters. A sample line of output, preceded by three blanks, might look like this:

```
027113776250010111O
```

3.
```
WRITEV (Final_Balance, OCT (Debits, 16), OCT (Credits, 16));
```

   The values of the variables Debits and Credits are converted to their octal equivalents and written to the string variable Final_Balance with 16 characters each. The output string, preceded by three blanks, might look like this:

```
'   77777770342      00000033766'
```

8.7.8 Prompting on Terminal Files

In VAX-11 PASCAL, if you open an interactive terminal file (such as OUTPUT) with the default carriage-control option LIST, you can use the WRITE procedure to prompt for input at the terminal. Each time you read from an interactive terminal file (such as 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 input on the same line. For example:

```
WRITE ('Name three presidents:');
READ (Pres1, Pres2, Pres3);
```

The system prints the prompt at the terminal, leaving the carriage positioned just after the colon (:). You can then begin typing input on the same line as the prompt. When the system executes the READ procedure, it finds the output string waiting to be printed.

Prompting works only for files associated with interactive terminals. For any other files, no output is written until the new line is started with a WRITELN. (Section 8.10 contains more information on prompting.)

Example

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

In this example, if the value of Num1 is 71.1 and the value of Num2 is 29.9, the resulting output to the terminal is:

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

Note that the chosen field width causes each of the real numbers to be preceded by a space.

8.8 Direct Access Procedures

The following procedures are generally legal only on files opened for direct access. In some cases, as indicated, the procedures apply to keyed access files as well.

- DELETE (also legal on files opened for keyed access)
- FIND
- LOCATE
- UPDATE (also legal on files opened for keyed access)

8.8.1 DELETE Procedure

The DELETE procedure deletes the current file component.

Syntax

DELETE (file-variable[, ERROR := error-recovery])

file-variable

The name of the file variable associated with the file from which a component is to be deleted.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the DELETE procedure is executing (see Section 8.2).

The file must be in Inspection mode before DELETE is called; the mode does not change after the procedure's execution.

When the DELETE procedure is called, the current component, as indicated by the file buffer, must have already been locked by a successful FIND, FINDK, GET, RESET, or RESETK procedure before it can be deleted. After deletion, the component is unlocked and UFB is TRUE.

DELETE can be used only on files with relative or indexed organization that have been opened for direct or keyed access; it cannot be used on files with sequential organization.

Example

```
DELETE (AccountsPayable);
```

This procedure call deletes the current component. When the component has been deleted, it is unlocked and UFB (AccountsPayable) is TRUE. A run-time error occurs if the current component of AccountsPayable is not locked.

8.8.2 FIND Procedure

The FIND procedure positions a file at a specified component.

Syntax

FIND (file-variable, component-number [, ERROR := error-recovery])

file-variable

The name of a file variable associated with a file that is open for direct access. The file must have fixed-length records.

component-number

A positive integer expression that indicates the component at which the file is to be positioned. If the component number is zero or negative, a run-time error occurs.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the FIND procedure is executing (see Section 8.2).

The FIND procedure allows direct access to the components of a file. You can use the FIND procedure to move forward or backward in a file.

The file must have been opened for direct access and may be in any mode before a call to FIND.

After execution of the FIND procedure, the file is positioned at the specified component. The file buffer variable assumes the value of the component, and the file mode is set to Inspection. If the file has relative organization, the current file component is locked. If there is no file component at the selected position, the file buffer is undefined (UFB becomes TRUE) and the mode becomes Undefined. After any call to FIND, the value of EOF is undefined.

You can use the FIND procedure only when reading a file that was opened by the OPEN procedure. If the file is open because of a default open (that is, with RESET or REWRITE), a call to FIND results in a run-time error because the default access method is sequential.

Examples

1. ```
   FIND (Albums, Current + 2);
   ```

   If the value of Current is 6, this procedure causes the file position to move to the eighth component; the file buffer variable Albums^ assumes the value of the component. If no eighth component exists, Albums^ is undefined and UFB (Albums) is TRUE.

2. ```
   FIND (Albums, Current - 1);
   ```

   If the value of Current is 6, this procedure causes the file position to move to the fifth component. The file buffer variable Albums^ assumes the value of the fifth component.

8.8.3 LOCATE Procedure

The LOCATE procedure positions a direct-access file at a particular component so that the next PUT procedure can modify that component.

Syntax

LOCATE (file-variable, component-number [, ERROR : error-recovery])

file-variable

The name of the file variable associated with the file to be positioned.

component-number

A positive integer expression that indicates the relative component number of the component to be found.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the LOCATE procedure is executing (see Section 8.2).

The file may be in any mode before LOCATE is called. The mode is set to Generation after the procedure's execution.

The LOCATE procedure positions the file so that the next PUT procedure will write the contents of the file buffer into the selected component. After LOCATE has been performed, UFB is TRUE and EOF is undefined.

Example

```
LOCATE (AccountsReceivable, 63);
AccountsReceivable^ := Next_Account;
PUT (AccountsReceivable);
```

The LOCATE procedure positions the file AccountsReceivable before relative component number 63. UFB (AccountsReceivable) is now TRUE and EOF (AccountsReceivable) is undefined. The assignment statement loads the file buffer with the contents of file position 63. The PUT operation writes the file buffer into file component number 63. UFB (AccountsReceivable) remains TRUE.

8.8.4 UPDATE Procedure

The UPDATE procedure writes the contents of the file buffer into the current component.

Syntax

UPDATE (file-variable[, ERROR := error-recovery])

file-variable

The name of the file variable associated with the file whose component is to be updated.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the UPDATE procedure is executing (see Section 8.2).

The file must be in Inspection mode before UPDATE is called; it remains in that mode after the procedure's execution.

The UPDATE procedure is legal only for files with relative or indexed organization that have been opened for direct or keyed access. The current component must have already been locked by a successful FIND, FINDK, GET, RESET, or RESETK procedure before the contents of the file buffer can be rewritten into it. After the update has taken place, the component is unlocked and UFB is TRUE.

Example

```
UPDATE (OctoberSales);
```

This procedure writes the file buffer contents (OctoberSales^) back into the current file component OctoberSales. The component is then unlocked and UFB (OctoberSales) is TRUE.

8.9 Keyed Access Procedures

The following procedures are legal only to files opened for keyed access.

- FINDK
- RESETK

8.9.1 FINDK Procedure

The FINDK procedure searches the index of an indexed file opened for keyed access and locates a specific component.

Syntax

FINDK (file-variable, key-number, key-value[[, match-type]]
[[, ERROR := error-recovery]])

file-variable

The name of the file variable associated with the file to be searched.

key-number

A positive integer expression that indicates the key position.

key-value

An expression that indicates the key to be found; it must be assignment compatible with the key field in the specified key position.

match-type

An identifier that indicates the relationship between the key value in the FINDK procedure call and the key value of a component.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the FINDK procedure is executing (see Section 8.2).

A component of an indexed file can have as many as 255 key fields. When you establish key fields with the KEY attribute (see Section 10.11), you assign each one a key number from 0 to 254. Key number 0 represents the mandatory primary key of the file. Separate indexes are built for each key number in the file.

The key value and the match type provide information about the key to be found. The key value must be assignment compatible with the key fields of the key number being searched. The match type must be one of the identifiers EQL, GTR, or GEQ, to indicate that the key to be found has a value equal to, greater than, or greater than or equal to the key value in the FINDK procedure call. The match type is optional; if omitted, it defaults to EQL.

The FINDK procedure can be called for any indexed file opened for keyed access, regardless of the file's mode. If the component described exists, the file buffer is filled with that component; UFB and EOF both become FALSE. The mode is set to Inspection and the component is automatically locked. If no component is found to match the description, UFB becomes TRUE and EOF is undefined. The mode is set to Undefined.

Example

```
FINDK (BookIndex, 1, 35, GEQ);
```

This procedure searches the index for key number 1 in the file BookIndex until it finds the first component whose key value is greater than or equal to 35. If the component matching the description in the FINDK statement is found,

UFB (BookIndex) and EOF (BookIndex) are FALSE, and the component is locked. If the component cannot be found, UFB (BookIndex) is TRUE, and EOF (BookIndex) is undefined. BookIndex must be an indexed file opened for keyed access.

8.9.2 RESETK Procedure

The RESETK procedure, like the RESET procedure described in Section 8.4.3, readies a file for reading.

Syntax

RESETK (file-variable, key-number[, ERROR := error-recovery])

file-variable

The name of the file variable associated with the input file.

key-number

A nonnegative integer expression that indicates the key position.

error-recovery

The parameter value that indicates the action to be taken if an error occurs while the RESETK procedure is executing (see Section 8.2).

The file can be in any mode before RESETK is called to set the mode to Inspection.

RESETK can be applied only to indexed files opened for keyed access. You assign a key number from 0 to 254 to each key field of a file component with the KEY attribute (see Section 10.11). The file is searched for the component with the lowest value in the specified key number. This component becomes the current component in the file and is locked. The value of the current component is copied into the file buffer; EOF and UFB are set to FALSE. If the component does not exist, EOF and UFB become TRUE. Note that a RESETK on key number 0 is equivalent to a RESET.

Example

```
RESETK (BookIndex, 0);
```

This procedure searches the file BookIndex for the component with the lowest value in the primary key. If this component exists, it becomes the current file component and is locked. UFB (BookIndex) and EOF (BookIndex) become FALSE. If the procedure was unable to find the component, UFB (BookIndex) and EOF (BookIndex) become TRUE. BookIndex must be an indexed file opened for keyed access.

8.10 Terminal I/O

The PASCAL language definition requires that the file buffer always contain the next file component that will be processed by the program. This definition can cause problems when the input to the program depends on the output most recently generated. To alleviate such problems in the processing of the text files, VAX-11 PASCAL uses a technique called delayed device access, also known as "lazy lookahead."

As a result of delayed device access, an item of data is not retrieved from a physical file device and inserted in the file buffer until the program is ready to process it. The file buffer is filled when the program makes the next reference to the file. A reference to the file consists of any use of the file buffer variable, including its implicit use in the GET, READ, and READLN procedures, or any test for the status of the file, namely, the EOF, EOLN, STATUS, and UFB functions.

The RESET procedure, which is required when any text file is opened for input, initiates the process of delayed device access. (Note that RESET is done automatically on the predeclared file INPUT.) RESET expects to fill the file buffer with the first component of the file. However, because of delayed device access, an item of data is not supplied from the input device to fill the file buffer until the next reference to the file.

When writing a program for which the input will be supplied by a text file, you should be aware that delayed device access occurs. Since RESET initiates delayed device access, and since EOF and EOLN cause the file buffer to be filled, you should place the first prompt for input before any tests for EOF or EOLN. The information you enter in response to the prompt supplies data that is retained by the file device until you make another reference to the input file.

Example

```
VAR
   I : INTEGER;
   .
   .
   .
BEGIN
WRITE ('Enter an integer or an empty line: ');
WHILE NOT EOLN DO
   BEGIN
   READLN (I);
   WRITELN ('The integer was: ', I:1);
   WRITE ('Enter an integer or an empty line: ');
   END;
WRITELN ('Done');
END.
```

The first reference to the file INPUT is the EOLN test in the WHILE statement. When the test is performed, the system attempts to read a line of input from the text file. Therefore, it is very important to prompt for the integer or empty line before testing for EOLN.

Suppose you respond to the first prompt by supplying an integer as input. Access to the input device is delayed until the EOLN function makes the first reference to the file INPUT. The EOLN function causes a line of text to be read into the internal line buffer. The subsequent READLN procedure reads the input value from the line of text and assigns it to the variable I. The WRITELN procedure writes the input value to the text file OUTPUT. The final statement in the WHILE loop is the request for another input value. The loop terminates when EOLN detects the end-of-line marker.

Delayed device access can produce unexpected results if you try to use the STATUS function to test the status of a text file after you have performed a READLN on the file. Remember that a READLN procedure call actually performs a READ procedure on each variable listed as a parameter, then performs a READLN to position the file at the beginning of the next line. Therefore, a call to STATUS after a READLN actually tests whether the file was successfully positioned. To test the status of the file, STATUS causes delayed device access to occur, thereby filling the file buffer with the next component. If you want to test the successful reading of data from the input file, you should read the data with the READ procedure, call the STATUS function, and then perform a READLN to advance the file to the beginning of the next line.

Chapter 9
Compilation Units

VAX-11 PASCAL includes two kinds of compilation units: programs and modules. Although both programs and modules have declaration sections, only programs have executable sections. A program can be compiled, linked, and executed by itself. A module, on the other hand, cannot be executed unless it is linked with a main program written in PASCAL or another language. (Note that although a module may contain routine declarations, these routines cannot be executed independently of a program.)

VAX-11 PASCAL gives you the option of writing modules that have the following characteristics:

- They can be combined with other separately compiled, but logically coordinated, programs and modules for execution as a single unit.
- They can be developed independently from other particular programs or modules, but used as library modules bound into larger systems at link time.

9.1 Compilation Unit Structure

VAX-11 PASCAL compilation units begin with a heading that identifies the program or module and lists the external file variables it uses.

Syntax

[[attribute-list]] {PROGRAM / MODULE} identifier [[({file-variable},...)]];

attribute-list

One or more identifiers that provide additional information about the compilation unit (see Chapter 10 for details).

identifier

The name of the program or module.

file-variable

The name(s) of the file variables associated with the external file(s) used by the compilation unit.

The identifier appears only in the heading and has no other purpose within the compilation unit. INPUT and OUTPUT must be listed in the heading if they are used. File variables for external files other than INPUT and OUTPUT must be listed in the heading and declared in the block. INPUT and OUTPUT should not be declared in the block. See Section 2.3.5 for more information on files.

The block of a program or module begins at the end of the heading and continues through the end of the compilation unit. In programs, the outermost block is divided into two sections: the declaration section and the executable section. In modules, the outermost block consists solely of a declaration section.

Examples

1. ```
   PROGRAM Test1;
   ```

   This program heading names the program Test1, but omits the file variable list; thus, this program does not use any external files.

2. ```
   MODULE Squares (INPUT, OUTPUT);
   ```

   This module heading names the module Squares and specifies the predeclared file variables INPUT and OUTPUT.

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

   This program heading names the program Payroll and specifies file variables for three external files: Employee, Salary, and OUTPUT. The files Employee and Salary must subsequently be declared in a VAR section of the program. Because OUTPUT is a predeclared file variable, it is not declared in the program.

9.2 Sharing Declarations and Definitions

By allowing compilation units to share declarations and definitions, VAX-11 PASCAL provides a means for separately compiled units to communicate with each other. The two sharing techniques supplied by VAX-11 PASCAL are:

- The use of global and external identifiers to share variables and routines (see Section 9.2.1). With this method, the identifiers are shared among all the compilation units that compose an executable image. This method is the only way for compilation units written in different languages to share declarations. The compiler does not check each declaration of an identifier to ensure that the identifier is always declared with the same type.
- The use of environment files to share variables, routines, constants, and types (see Section 9.2.2). With this method, declarations and definitions are shared only among those compilation units that "inherit" them. The compiler checks to make sure that every use of a shared object is legal for an object of its type. This method can be used only when all the compilation units involved are written in PASCAL.
   ```

9.2.1 Using Global and External Identifiers

Variables and routines declared in a compilation unit can be referred to in another compilation unit if the declaration in the first compilation unit includes the GLOBAL attribute, and the declaration in the second compilation unit includes the EXTERNAL attribute (see Section 10.20 for descriptions of both attributes). Because the compiler performs no type checking in this case, to avoid errors you must make sure that both declarations specify the same type.

You cannot use global and external names to share the definitions of symbolic constants and user-defined types.

For example, assume program A and module B share identifiers as follows:

File A.PAS

```
PROGRAM A (INPUT, OUTPUT);

CONST
   Rate = 0.06;

VAR
   Amt, Total, Tax: [GLOBAL] REAL;

[EXTERNAL] PROCEDURE Calc;
   EXTERN;

[GLOBAL] PROCEDURE Glf;

   BEGIN
      .
      .
      .
   END;

BEGIN
   .
   .
   .
READ (Amt);
Calc;
WRITELN ('PURCHASE AMOUNT ', Amt:10:2);
WRITELN ('               + ', Tax:10:2);
WRITELN ('PAY THIS TOTAL   ', Total:10:2);
END.
```

File B.PAS

```
MODULE B;

CONST
   Rate = 0.06;

VAR
   Amt, Total, Tax: [EXTERNAL] REAL;

[EXTERNAL] PROCEDURE Glf;
   EXTERN;

[GLOBAL] PROCEDURE Calc;

   BEGIN
   Tax := Amt * Rate;
   Total := Tax + Amt;
   Glf;
      .
      .
      .
   END;
END.
```

In program A, the GLOBAL attribute specifies that Amt, Total, and Tax are global variables. Module B can refer to Amt, Total, and Tax because it uses the EXTERNAL attribute to specify that these variables were declared in another compilation unit. Amt, Total, and Tax must be declared to be of the same type in both compilation units. Similarly, Calc and Glf are declared as global procedures in one compilation unit and as external procedures in the other. The result is that each compilation unit can call the other's procedures.

9.2.2 Using Environment Files

The ENVIRONMENT and INHERIT attributes (described in Sections 10.7 and 10.9) allow programs and modules to share the definitions of symbolic constants and user-defined types and the declarations of variables and routines from previously compiled units. See Chapter 10 for descriptions of attributes other than ENVIRONMENT and INHERIT that apply to compilation units.

An environment consists of descriptions of constant, type, variable, procedure, and function identifiers declared at the outermost level of a compilation unit. When one compilation unit inherits the environment of another, the effect is to incorporate the environment file directly into the first compilation unit.

An environment file is similar to a %INCLUDE file, with the following differences:

- An environment file can contain only declarations and definitions.
- The environment file and the compilation unit that inherits it are checked to ensure that their versions are consistent (see the *VAX-11 PASCAL User's Guide* for details on version consistency).

- The data in the environment file exists in a form that the compiler can handle more easily.
- A variable inherited from an environment file is not a newly created variable, but is instead the same variable that was allocated storage by the declaring compilation unit.

The following sections describe the uses of the ENVIRONMENT and INHERIT attributes. Section 9.2.2.3 explains the rules governing multiple declarations of identifiers.

9.2.2.1 ENVIRONMENT Attribute — To define the environment of a compilation unit, include the ENVIRONMENT attribute and a VAX/VMS file specification (enclosed in apostrophes) in an attribute list immediately preceding the program or module heading. The declarations and definitions made at the outermost level of the compilation unit are saved in a file identified by the file specification.

For example, when the following program is compiled, an environment file named CALC.PEN is created:

```
[ENVIRONMENT('CALC.PEN')] PROGRAM Calc (INPUT, OUTPUT);

LABEL 75;

CONST
   Pi = 3.1415927;

TYPE
   Yes_No = (Yes, No);

VAR
   Operand : REAL;
   Subtotal : REAL := 0.00;
   Operator : CHAR;
   Answer : Yes_No;

PROCEDURE Instructions;
   .
   .
   .
```

Descriptions of Pi, Yes_No, Subtotal, Operand, Operator, Answer, and Instructions are included in CALC.PEN. Environment files do not include labels or variable initializations; therefore, the label 75 and the initialization of Subtotal to 0.00 are not part of CALC.PEN.

9.2.2.2 INHERIT Attribute — Once an environment has been defined, other modules can reference the identifiers it declares by inheriting the environment with the INHERIT attribute. For example, if you include INHERIT ('CALC.PEN') in the attribute list immediately preceding a module heading, that module can refer to Pi, Yes_No, Subtotal, and so on, just as if the identifiers had been declared at the outermost level of the module itself.

Using environment files, you would write program A and module B from Section 9.2.1 as follows:

File A.PAS

```
[ENVIRONMENT('A.PEN')] PROGRAM A (INPUT, OUTPUT);

CONST
   Rate = 0.06;

VAR
   Amt, Total, Tax : REAL;

PROCEDURE Calc;
   EXTERNAL;

PROCEDURE Glf;

   BEGIN
      .
      .
      .
   END;

BEGIN
   .
   .
   .
READ (Amt);
Calc;
WRITELN ('PURCHASE AMOUNT ', Amt:10:2);
WRITELN ('               + ', Tax:10:2);
WRITELN ('PAY THIS TOTAL   ', Total:10:2);
END.
```

File B.PAS

```
[INHERIT('A.PEN')] MODULE B;

[GLOBAL] PROCEDURE Calc;

   BEGIN
   Tax := Amt * Rate;
   Total := Tax + Amt;
   Glf;
      .
      .
      .
   END;
END.
```

The ability to share environments allows module B to eliminate the CONST definition, the VAR declarations, and the declaration of the procedure Glf. Program A does not inherit the environment of module B; therefore, the procedure Calc still must be declared GLOBAL in the called program (module B) and EXTERNAL in the calling program (program A).

A compilation unit can define and inherit any number of environments. However, each file specification associated with an INHERIT attribute must represent an environment file created by an earlier compilation. The identifiers inherited by the compilation unit are not included in the environment defined

by the unit. In the following example, the environment created by the compilation of ModA includes the declarations from the outermost level of ModA, but NOT those from Main and ModB:

```
[ENVIRONMENT('MAIN.PEN')] PROGRAM Main;

[INHERIT('MODB.PEN'), ENVIRONMENT('MODA.PEN')] MODULE ModA;

[INHERIT('MAIN.PEN'), ENVIRONMENT('MODB.PEN')] MODULE ModB;
```

9.2.2.3 Multiply Declared Names — The identifiers to which you can refer at the outermost level of a compilation unit—that is, all those defined in the outermost level of the compilation unit itself, plus those declared in all inherited environments—must be unique. Thus, the same identifier cannot be declared in two simultaneously inherited environments, and an identifier inherited from an environment cannot be redeclared at the outermost level of the compilation unit.

Several exceptions to this redeclaration rule are needed, because a module can inherit the environment of a program that calls its global procedures or functions, or refer to its global variables. Such a conflict occurs in the compilation units in the following example:

File PROG.PAS

```
[ENVIRONMENT('PROG.PEN')] PROGRAM Prog;

[EXTERNAL] PROCEDURE Inst;
   EXTERN;
      .
      .
      .
BEGIN
   .
   .
   .
Inst;
   .
   .
   .
END.
```

File MOD.PAS

```
[INHERIT('PROG.PEN')] MODULE Mod;

[GLOBAL] PROCEDURE Inst;

   BEGIN
      .
      .
      .
   END;
END.
```

The procedure Inst is defined in Mod and called in the executable block of Prog. Mod inherits the environment created by the compilation of Prog; thus, the identifier Inst is said to be multiply declared in Mod.

The same problem could occur with global functions or variables. Therefore, VAX-11 PASCAL allows the following exceptions to the redeclaration rule:

- A variable identifier may be multiply declared if all declarations of the variable have the same type, and all but one declaration at most are external.
- A procedure identifier may be multiply declared if all declarations of the procedure have congruent parameter lists (see Section 6.6.6), and all but one declaration at most are external.
- A function identifier may be multiply declared if all declarations of the function have congruent parameter lists and identical result types, and all but one declaration at most are external.

If one declaration of a variable or a routine is not external, it must be a global declaration.

9.2.3 Examples

1.
```
[ENVIRONMENT('MOD1.PEN')] MODULE Mod1;

[ENVIRONMENT('MOD2.PEN')] MODULE Mod2;

[ENVIRONMENT('MOD3.PEN')] MODULE Mod3;

[INHERIT('MOD1.PEN','MOD2.PEN','MOD3.PEN')] PROGRAM Prog;
```

   This example shows how large systems can be split into several functional components, or modules, that share environments. These four source files are equivalent to one long PASCAL program that includes all the declarations defined in all the modules. A modular design allows you to treat different parts of a system individually; you can develop the components separately, and later, when changes are needed, you can update and recompile one module without having to recompile them all.

2.
```
[ENVIRONMENT('NEWWRITE.PEN')] MODULE Newwrite;
```

   This example shows the heading of a module called Newwrite, which might contain a special output routine. A compilation unit that wanted to use this routine could inherit the module's environment as follows:

```
[INHERIT('NEWWRITE.PEN')] MODULE Process;
```

   The module Process now has access to all the definitions and declarations in Newwrite. The environment for Newwrite can define the special symbolic constants and user-defined types needed to call the new routine. The availability of these additional symbolic constants and user-defined types enables Process to communicate easily and efficiently with the called routine.

Chapter 10
Attributes

An attribute is an identifier that directs the VAX-11 PASCAL compiler to change its behavior in some way. Attributes allow additional control over the properties of data items, routines, and compilation units. An attribute class can consist of a single attribute identifier, or of several attribute identifiers with a common characteristic. When an attribute is not explicitly stated, the compiler follows default rules to assign properties to program elements. Table 10–1 lists the attribute classes that can be applied to formal routine parameters, routines, and compilation units. Table 10–2 lists the attribute classes that can be applied to data items.

Table 10–1: Attributes on Routines and Compilation Units

Class	Program Element		
	Routine Parameter	Routine	Compilation Unit
Allocation	No	Yes[1]	Yes[1]
ASYNCHRONOUS	Yes	Yes	No
CHECK	No	Yes	Yes
Double-Precision	No	No	Yes
ENVIRONMENT	No	No	Yes
IDENT	No	No	Yes
INHERIT	No	No	Yes
INITIALIZE	No	Yes	No
LIST	Yes	No	No
Optimization	No	Yes	Yes
OVERLAID	No	No	Yes
UNBOUND	Yes	Yes	No
Visibility	No	Yes	Yes[2]

1. PSECT is the only allocation attribute allowed
2. EXTERNAL and WEAK_EXTERNAL not allowed

Table 10–2: Attributes on Data Items

Class	Data Item					
	Variable	Formal Parameter	Pointer Base Type	Component[1]	Function Result	Various Items[2]
Alignment	Yes	Yes[3]	Yes[3]	Yes[4]	Yes	No
Allocation	Yes	No	No	No	No	No
KEY	No	No	No	Yes[5]	No	No
LIST	No	Yes	No	No	No	No
POS	No	No	No	Yes[5]	No	No
READONLY	Yes	Yes	Yes	Yes	No	No
Size	Yes	Yes[6]	Yes	Yes[7]	Yes	No
UNSAFE	Yes	Yes[8]	Yes	Yes	Yes	Yes
Visibility	Yes	No	No	No	No	No
VOLATILE	Yes	Yes	Yes	Yes	Yes	No
WRITEONLY	Yes	Yes	Yes	Yes	No	No

1. Component of a record, array, VARYING string, or file (includes conformant schemes)
2. Index of an array, tag field of a variant record (when no tag identifier is present), base type of a set
3. UNALIGNED not allowed
4. Not allowed on components of files or VARYING strings
5. Allowed only on record fields (including the tag field of a variant record)
6. Not allowed on conformant parameters
7. Not allowed on components of files or VARYING strings, or on structured types with file components
8. Not allowed on conformant VARYING parameters

10.1 Specifying Attributes

A list of attributes enclosed in brackets can appear anywhere in a program that a type, a type identifier, or the heading of a routine or compilation unit is allowed. However, only one attribute from a particular class can appear in a given attribute list. The use of attribute lists is illustrated in the appropriate syntax diagrams throughout this manual. Notice that the names of attributes, when used in a suitable context, do not conflict with other identifiers with the same name in the program.

Syntax

identifier1

The name of the attribute.

constant-expression

A compile-time integer expression, represented in this chapter by n, that qualifies several of the VAX-11 PASCAL attributes.

identifier2

The name of an option available with the CHECK attribute or of a storage area indicated by the COMMON and PSECT attributes.

Some attributes require a special form of constant expression called a name-string. The syntax of a name-string differs from that of other strings in VAX-11 PASCAL in that a name-string cannot use the extended string syntax (see Section 2.3.2.2).

Every program element listed in Tables 10–1 and 10–2 must be associated with one property for which there is an applicable attribute class. If the program does not give each program element an explicit attribute from each class, the VAX-11 PASCAL compiler automatically supplies the defaults for the unspecified classes at the time of the element's declaration. In some classes, as described in the following sections, the default property is not available through an explicit attribute.

Attributes can be associated with data items in two ways:[1]

- By appearing in a type definition in a TYPE section; the item is later declared to be of that type.
- By appearing in the declaration of an item preceding its type.

When a type definition includes a list of attributes, the type has only those attributes specified. The compiler does not supply the defaults for the unspecified classes until a data item is declared to be of that type. Two rules govern the use of attributes in a TYPE section:

- The attributes of the type can neither conflict with nor duplicate any attributes explicitly stated in the data item's declaration.
- The type cannot be used anywhere that its accompanying attributes are illegal.

The following examples show both legal and illegal uses of attributes in type definitions:

```
TYPE
   A = [GLOBAL] INTEGER;
   B = [UNALIGNED] INTEGER;

VAR
   A1 : [GLOBAL] A;          (* Illegal; duplicates
                                GLOBAL attribute of
                                type A *)
   A2 : [EXTERNAL] A;        (* Illegal; conflicts with
                                GLOBAL attribute of
                                type A *)
   B1 : ^B;                  (* Illegal; pointer base
                                type cannot be
                                UNALIGNED *)
   C : A;                    (* Legal *)
```

1. The presence in VAX-11 PASCAL of compile-time expressions and attribute lists leads to a minor ambiguity in the language syntax. If the compiler finds a left bracket ([) symbol when it expects to find a type or type identifier, it always assumes that the bracket indicates the beginning of an attribute list. The ambiguity arises because the left bracket could also represent the beginning of a set constructor that denotes the low bound of a subrange type. If the latter case is in fact what you intend, simply parenthesize the set constructor; the compiler will interpret the expression correctly.

The first three variable declarations are illegal for the reasons shown in the comments. The declaration of C is legal; C is declared as a GLOBAL integer variable because of the characteristics of its type. The compiler supplies defaults for all other classes applicable to the variable C.

Attributes associated with data items usually modify type compatibility rules. The sections of this chapter pertaining to the accessibility, alignment, ASYNCHRONOUS, LIST, POS, size, UNBOUND, UNSAFE, and VOLATILE attributes describe their effects on type compatibility. Attributes applied to components of structured types affect the entire structure. The sections discussing the accessibility, alignment, size, and volatility attributes also present the rules for using these attributes with structured types.

The following sections describe the attribute classes in alphabetical order. Note that in this chapter, the term "object" is used to indicate any program element to which the attributes of the class can be applied.

10.2 Alignment Attributes

The alignment attributes can be applied to variables, the base types of pointer variables, components of structured variables, and function results. They indicate whether the object should be aligned on a specific addressing boundary in memory.

ALIGNED [[(n)]]

An ALIGNED object is aligned on the memory boundary indicated by n. The constant expression n indicates that the address of the object must end in at least n zeros. ALIGNED(0) specifies byte alignment, ALIGNED(1) specifies word alignment, ALIGNED(2) specifies longword alignment, ALIGNED(3) specifies quadword alignment, ALIGNED(4) specifies octaword alignment, and ALIGNED(9) specifies page alignment.

UNALIGNED

An UNALIGNED object may be aligned on any bit boundary.

Rules and Defaults

- The default alignment of an object depends on its size. The *VAX-11 PASCAL User's Guide* contains the complete rules for default alignment.
- In VAX-11 PASCAL, an UNALIGNED variable cannot have an allocation size greater than 32 bits.
- The constant expression n must denote an integer. If you omit it, the default is 0, indicating byte alignment.
- ALIGNED(9) is the largest alignment allowed.
- An AUTOMATIC variable (see Section 10.3) cannot have alignment greater than a longword.
- The minimum alignment for an object of a structured type is the greatest alignment specified for any of its components.

- Alignment attributes are illegal on components of files and VARYING strings.
- The alignment of a formal VAR parameter cannot be greater than the alignment of a corresponding actual parameter, either by default or by means of an alignment attribute. In an array variable passed to a conformant formal parameter, alignment and size attributes (see Section 10.17) are illegal on all dimensions of the actual parameter, except the first, that correspond to the dimensions of the formal parameter.
- A formal parameter cannot be UNALIGNED. Thus, an UNALIGNED variable cannot be passed to a formal VAR parameter.
- The base type of a pointer variable passed to the NEW procedure cannot have alignment greater than a quadword, nor can it be UNALIGNED.
- If the base type of a pointer variable has a specified alignment, then the base type of a pointer expression assigned to it must have an alignment equal to that of the variable.
- Pointer types are structurally compatible only if their base types have identical alignment.

Example

```
VAR
    Free_Buffers : [ALIGNED(1),WORD] -2**15..2**15-1;
    .
    .
    .
IF ADD_INTERLOCKED (-1, Free_Buffers) <= 0
THEN
    .
    .
    .
```

The predeclared function ADD_INTERLOCKED requires that the second parameter passed to it have word alignment and an allocation size of one word. In this example, the variable Free_Buffers is declared with alignment and size attributes to meet these restrictions. (The ADD_INTERLOCKED function is described in Section 7.9.1.)

10.3 Allocation Attributes

The allocation attributes can be applied to variables, routines, and compilation units. They indicate the form of storage that the object should occupy.

STATIC

Storage for a STATIC variable is allocated only once. A STATIC variable exists as long as the executable image in which it was allocated remains active.

AUTOMATIC

Storage for an AUTOMATIC variable is allocated each time the program enters the routine in which the variable was declared. The storage is deallocated each time the program exits from that routine. An AUTOMATIC variable exists as long as the declaring routine remains active.

AT(n)

No storage is allocated for a variable having the AT attribute. The variable is assumed to reside at the exact address specified by the constant expression n. Variables representing machine-dependent entities are fre quently given the AT attribute.

COMMON [[(identifier)]]

Storage for a variable having the COMMON attribute is allocated in an overlaid program section called a common block. This attribute allows you to share variables with other languages (such as FORTRAN). If you include an identifier in the attribute, it indicates the name of the common block. If you omit the identifier, the name of the variable is used as the name of the common block. See the *VAX-11 PASCAL User's Guide* for details.

PSECT(identifier)

The identifier designates the program section in which storage for an object is to be allocated. Storage for the object remains allocated as long as the executable image in which the object was declared remains active. See the *VAX-11 PASCAL User's Guide* for details.

Rules and Defaults

- PSECT is the only allocation attribute that can be applied to routines and compilation units.
- By default, variables declared in nested blocks are automatic.
- By default, variables declared at the outermost level of a module are static.
- By default, variables declared at the outermost level of a program are static, although for efficiency they may actually be made automatic (see the *VAX-11 PASCAL User's Guide*.
- Program-level variables with the AUTOMATIC attribute are not recorded in environment files.
- GLOBAL and EXTERNAL variables (see Section 10.20) are implicitly static. Thus, they conflict with the AUTOMATIC attribute.
- A variable having the AT, COMMON, or PSECT attribute is implicitly static.
- The COMMON attribute can be applied only to variables.
- Only one variable can be allocated in a particular common block. Therefore, the name of the common block cannot be used as the name of another common block or program section.
- If a PASCAL program shares a record variable with a FORTRAN program, the fields must be laid out identically in both common blocks.

Example

```
PROGRAM Print_Random (OUTPUT);

VAR
   I : [AUTOMATIC] INTEGER;

FUNCTION Random
   : INTEGER;

   VAR
      X : [STATIC] INTEGER := 15;

   BEGIN
   X := ((9 * X) + 7) MOD 11;
   Random := X;
   END;

BEGIN
FOR I := 1 TO 20 DO
   WRITELN (Random);
END.
```

The program Print_Random includes a function that generates a random integer. Because the variable X is declared STATIC, its value will be preserved from one activation of the function to the next. By default, the storage for X would have been deallocated when control returned to the main program. Because X is STATIC, it retains the value it had when Random ended and assumes this value the next time Random is called. In the program Print_Random, the program-level variable I is declared AUTOMATIC to override the default static allocation.

10.4 ASYNCHRONOUS Attribute

The ASYNCHRONOUS attribute can be applied to routines and routine parameters declared in external routines to indicate that the routine may be called by an asynchronous event. Since such an event can alter the values of variables within the routine unpredictably, the ASYNCHRONOUS attribute changes how the routine is optimized.

Rules and Defaults

- In the absence of an ASYNCHRONOUS attribute, the compiler assumes that the routine can be activated only by actual calls within the program.
- All predeclared routines are ASYNCHRONOUS by default.
- Any routines called from within the block of an ASYNCHRONOUS routine must be local to the ASYNCHRONOUS routine or must be themselves ASYNCHRONOUS, either by default or by an explicit attribute.
- All nonlocal variables accessed from within the block of an ASYNCHRONOUS routine must be declared VOLATILE (see Section 10.21).
- If a formal routine parameter is ASYNCHRONOUS, all actual parameters passed to it must also be ASYNCHRONOUS.
- An ASYNCHRONOUS routine may be passed as an actual parameter to a formal routine parameter that does not have this attribute.

Example

```
PROCEDURE Do_Something;

   VAR
      I : [VOLATILE] INTEGER;
      J : INTEGER

   [ASYNCHRONOUS] FUNCTION Handler
      : BOOLEAN;

      BEGIN
          .
          .
          .
      I := I + 1;
          .
          .
          .
      END;

   BEGIN
   ESTABLISH (Handler);
      .
      .
      .
   END;
```

This example illustrates the declaration of an ASYNCHRONOUS function, Handler. Note that the executable section of Handler cannot access variables declared in the enclosing block of the procedure Do_Something unless those variables are declared VOLATILE. Thus, Handler can access the variable I, which has the VOLATILE attribute, but cannot access the variable J.

10.5 CHECK Attribute

The CHECK attribute can be applied to routines and compilation units. It specifies error-checking options that are to be enabled during program execution.

CHECK([identifier],...)

The options listed with the CHECK attribute are enabled. If you omit the list of options, all available positive options are enabled.

The options listed in Table 10–3 allow you to choose which aspects of a program should be checked. Options enable the specified checking features while their negations disable them. For example, the POINTERS option checks the addresses to which pointer variables refer. If you specify the NO-POINTERS option for a routine, the checking of pointer addresses is disabled inside that block.

Table 10–3: Summary of Checking Options

Option	Action	Negation
ALL	Enables all forms of checking	NONE
BOUNDS	Verifies that an index expression is within the bounds of an array's index type and that character-string sizes are compatible with the operations being performed	NOBOUNDS
CASE_SELECTORS	Verifies that the value of a case selector is contained in the corresponding case label list	NOCASE_SELECTORS
OVERFLOW	Verifies that the result of an integer computation does not exceed the machine representation	NOOVERFLOW
POINTERS	Verifies that the value of a pointer variable is not NIL	NOPOINTERS
SUBRANGE	Verifies that values assigned to variables of subrange types are within the subrange; verifies that a set expression is assignment compatible with a set variable	NOSUBRANGE

Rules and Defaults

- BOUNDS is the only option enabled by default. The defaults for the other options are NOCASE_SELECTORS, NOOVERFLOW, NOPOINTERS, and NOSUBRANGE. If you wish to enable any of the options, you must specify them with the CHECK attribute.

Example

```
PROGRAM Check_Features;

[CHECK(POINTERS,CASE_SELECTORS)] PROCEDURE Linked_List
    (VAR Client : Info_Rec);
    .
    .
    .
[CHECK(OVERFLOW)] FUNCTION Integer_Compute
    (VAR Int1, Int2, Int3 : INTEGER)
    : INTEGER;
    .
    .
    .
PROCEDURE Bounds_Check
    (VAR String : VARYING[30] OF CHAR;
    VAR Char_Array : ARRAY[1..25] OF CHAR;
    VAR Half_Alpha : 'A'..'M');
    .
    .
    .
```

The routines Linked_List and Integer_Compute will have the specified options plus the BOUNDS option enabled (by default). All other options will remain disabled when Linked_List and Integer_Compute are called. The procedure Bounds_Check will have only the default BOUNDS option enabled.

10.6 Double-Precision Attributes

Double-precision attributes can be applied to compilation units to indicate which format should be used to represent double-precision real numbers. These attributes choose the internal hardware representation to be used for items of type DOUBLE within the compilation unit. See Section 2.2 for a discussion of the two types of double-precision real numbers; see the *VAX-11 PASCAL User's Guide* for details of the hardware representation.

G_FLOATING

Double-precision variables and expressions in the compilation unit are represented in G_floating format. Their values have an approximate range from 10**-308 through 10**308 and an approximate precision of 15 decimal digits. Not all VAX-11 processors support the G_floating data type.

NOG_FLOATING

Double-precision variables and expressions in the compilation unit are represented in D_floating format. Their values have an approximate range from 10**38 through 10**38 and an approximate precision of 16 decimal digits.

Rules and Defaults

- NOG_FLOATING is the default double-precision attribute.
- All independently compiled units that are linked together should use the same double-precision format.

Example

```
[G_FLOATING,ENVIRONMENT('REALDATA.PEN')] MODULE Real_Data;

[G_FLOATING,ENVIRONMENT('STRINGDATA.PEN')] MODULE String_Data;

[G_FLOATING,INHERIT('REALDATA.PEN','STRINGDATA.PEN')]
   PROGRAM Record_Keeping;
```

This example shows the headings of a program and the two modules whose environments it inherits. Note that all three compilation units must specify the G_FLOATING attribute in order for the G_floating format of representation to be used.

10.7 ENVIRONMENT Attribute

The ENVIRONMENT attribute can be applied to compilation units and causes the unit's program- or module-level declarations and definitions to be saved.

ENVIRONMENT(name-string)

The declarations and definitions made at the outermost level of the compilation unit (provided they do not have the AUTOMATIC attribute) are saved in a newly created environment file. You must name this file by including a VAX/VMS file specification in the name-string (see Section 10.1 for the syntax of a name-string). The ENVIRONMENT attribute makes the contents of an environment file available to other compilation units that inherit it. See Chapter 9 for further explanation and examples.

10.8 IDENT Attribute

The IDENT attribute can be used to qualify the name of a compilation unit. See the *VAX-11 PASCAL User's Guide* for a description of this attribute.

IDENT(name-string)

The name-string can contain additional information whose use is implementation specific. The VAX-11 PASCAL compiler uses this string to supply identification information to the linker. (See Section 10.1 for the syntax of a name-string.)

Rules and Defaults

- In the absence of an IDENT attribute, the string '01' is supplied to the linker.

10.9 INHERIT Attribute

The INHERIT attribute can be used to indicate the environment files to be inherited by a compilation unit.

INHERIT({name-string},...)

The compilation unit inherits one or more environment files named by the VAX/VMS file descriptions in the name-strings (see Section 10.1 for the syntax of a name-string). These files contain declarations and definitions made at program or module level in other compilation units. See Chapter 9 for further explanation and examples.

Rules and Defaults

- The environment files specified by the INHERIT attribute must have already been created in compilation units that have the ENVIRONMENT attribute.

10.10 INITIALIZE Attribute

The INITIALIZE attribute can be applied to procedures to indicate that the procedure is to be called before the main program is entered. A compilation unit may include any number of INITIALIZE procedures, all of which are called in an unspecified order before the main program is entered.

Rules and Defaults

- In the absence of an INITIALIZE attribute, the compiler assumes that a routine can be activated only by actual calls within the program.
- By default, INITIALIZE procedures have the characteristics of UNBOUND routines (see Section 10.18).
- An INITIALIZE procedure cannot have a formal parameter list.
- An INITIALIZE procedure cannot be external.

Example

```
PROGRAM Routine_Activate;

[INITIALIZE] PROCEDURE Check_Open;
    .
    .
    .
BEGIN (* Routine_Activate *)
    .
    .
    .
```

In this example, the body of the INITIALIZE procedure Check_Open will be executed before the main program is activated.

10.11 KEY Attribute

The KEY attribute can be applied to record fields. KEY indicates that the field is to be used as a key field when the record is part of an indexed sequential file.

KEY [[(n)]]

A key number of 0 indicates that the field is the primary key of the record. All other key numbers indicate alternate keys.

Rules and Defaults

- The key number n must be a constant expression that denotes an integer value in the range from 0 through 254. If you omit the key number, the default value is 0.
- When you create a new indexed file with more than one key field, you must make sure that the defined keys are dense; that is, you may not omit any key numbers in the range from 0 through the highest key number specified.
- The KEY attribute is ignored except when the record is the component type of a file (see Chapter 8).

- A key field can be of any ordinal type or of type PACKED ARRAY OF CHAR. If the key field is of type PACKED ARRAY OF CHAR, its length cannot exceed 255 characters.
- The KEY attribute does not affect type compatibility rules.
- A key field may not be UNALIGNED (see Section 10.2).
- A key field of an ordinal type must be allocated exactly one byte, one word, or one longword. This restriction is imposed by VAX-11 RMS.
- An integer key field that is allocated one byte may not have negative values. This restriction is imposed by VAX-11 RMS.

Example

```
TYPE
   Register = RECORD
              Student_No : [KEY(0)] INTEGER;
              Student_Name : RECORD
                             Last_Name : PACKED ARRAY[1..20]
                                         OF CHAR;
                             First_Name : PACKED ARRAY[1..15]
                                          OF CHAR;
                             Initial : CHAR;
                             END;
              Course_Load : INTEGER;
              Grade_Average : REAL;
              Class : [KEY(1)] PACKED ARRAY[1..9] OF CHAR;
              END;
```

This example defines the identifier Register to denote a record type. The first field, Student_No, is the primary key of the record. Register contains another field, Class, which is established as the alternate key.

10.12 LIST Attribute

The LIST attribute can be applied to a formal parameter of a routine not written in PASCAL. LIST indicates that the routine may be called with multiple actual parameters that correspond to the last formal parameter named in the routine heading.

Rules and Defaults

- In the absence of a LIST attribute, an error results if the number of actual parameters exceeds the number of formal parameters.
- The LIST attribute can be applied only to the last formal parameter in a parameter list.
- You may supply zero, one, or more than one actual parameter to correspond to a LIST formal parameter, but you must use positional syntax when supplying them. The number of actual parameters you can supply is limited by VAX/VMS to 255.
- You may use the LIST attribute on procedure and function parameters to indicate that an external routine can take an arbitrary number of routine parameters.

- All actual parameters that correspond to a LIST formal parameter must be compatible (or congruent) with the type of the formal parameter.
- For formal and actual parameter lists to be congruent, both the actual routine parameter and the corresponding formal routine parameter must either have the LIST attribute or lack the LIST attribute.

Example

```
PROGRAM Arg_Mech;

[EXTERNAL(MTH$JMAX0)] FUNCTION JMax0
   (Int_List : [LIST] INTEGER)
   : INTEGER;
   EXTERN;

VAR
   I, J, K, L : INTEGER;
   Int_Array : ARRAY[1..10] OF INTEGER;

BEGIN (* Main Program *)
   .
   .
   .
I := JMax0 (J, K, L, Int_Array[J+1], Int_Array[K+2],
   Int_Array[L+3]);
END.
```

The program Arg_Mech illustrates the effect of the program, this routine is known as the function JMax0. JMax0 is declared with one formal LIST parameter; therefore, the function designator in this example contains excess actual parameter entries. Any number of integer expressions can be passed as actual parameters when JMax0 is called.

10.13 Optimization Attributes

Optimization attributes can be applied to routines and compilation units to indicate whether the VAX-11 PASCAL compiler should optimize code. See the *VAX-11 PASCAL User's Guide* for more information on optimization.

OPTIMIZE

The compiler is allowed to optimize the code for the object.

NOOPTIMIZE

The compiler is prohibited from optimizing the code for the object. The NOOPTIMIZE attribute ensures that expressions will be evaluated completely, from left to right.

Rules and Defaults

- OPTIMIZE is the default optimization attribute.

Example

```
PROGRAM Numbers;

[NOOPTIMIZE] PROCEDURE Process_Negative;
    .
    .
    .
```

This example shows the use of the NOOPTIMIZE attributes to disable optimization of the code for the routine Process_Negative. Code for the rest of the program will be optimized.

10.14 OVERLAID Attribute

The OVERLAID attribute can be applied to compilation units to indicate how storage should be allocated for variables declared within the unit. If you specify OVERLAID, the variables declared at program or module level (unless they have the STATIC or PSECT attribute) will overlay the storage of static variables in all other OVERLAID compilation units. See the *VAX-11 PASCAL User's Guide* for more information.

This attribute is intended for use only with programs that use the decommitted separate compilation facility provided by Version 1 of VAX-11 PASCAL.

Rules and Defaults

- By default, variables are not stored in OVERLAID compilation units.

Example

```
[OVERLAID] PROGRAM A;

[OVERLAID] PROGRAM B;
```

Because the OVERLAID attribute is specified, the variables declared at the outermost level of program B will overlay those declared at the outermost level of program A.

10.15 POS Attribute

The POS attribute can be applied to a field of a packed or an unpacked record. POS forces the field to a specific bit position within the record.

POS(n)

The constant expression n specifies the bit location, relative to the beginning of the record, at which the field begins.

Rules and Defaults

- VAX-11 PASCAL's defaults for the positioning of record fields are described in the *VAX-11 PASCAL User's Guide.*
- The constant expression n cannot denote a negative integer.
- The beginning position of a field must be greater than the ending position of the field preceding it.

- Inside a record variant, the beginning position of a field must be greater than the ending position of the preceding field within the same variant. As always, the variants themselves may overlap.
- A record variable containing a field of a file type cannot include a POS attribute for any field.
- In VAX-11 PASCAL, a field whose allocation size is greater than 32 bits must be positioned on a byte boundary.
- The specified bit position must not conflict with the alignment explicitly required by an alignment attribute (see Section 10.2).
- Two record types in which corresponding fields are not identically positioned are neither assignment compatible nor structurally compatible.

Example

```
TYPE
   Control = RECORD
             Flag_1 : [BIT,POS(0)] BOOLEAN;
             Flag_2 : [BIT,POS(1)] BOOLEAN;
             Count : [BYTE,ALIGNED] 0..100;
             Error : [BIT,POS(31)] BOOLEAN;
             END;
```

This example uses the POS attribute to position the fields of an unpacked record such that Flag_1 occupies bit 0, Flag_2 occupies bit 1, and Error occupies bit 31. Because the Count field has size and alignment attributes, it is allocated one byte of storage and is aligned on the byte boundary following Flag_2; that is, storage for Count occupies bits 8 through 15. Bits 2 through 7 and 16 through 30 are left empty; there is no way for you to refer to them.

10.16 READONLY Attribute

The READONLY attribute can be applied to variables, formal parameters, the base types of pointer variables, and components of structured variables. READONLY specifies that an object can be read by a program but cannot have values assigned to it. For example, if A is a READONLY formal parameter, you can use it in an expression such as C := A + B. You can also use A as a value parameter in routine calls such as ORD (A), and you can pass A as a READONLY VAR parameter. You cannot, however, assign values to A, as in A := B + C.

Rules and Defaults

- By default, an object can be both read and written.
- No value of any type is assignment compatible with a READONLY object.
- The presence of a READONLY component in an object of a structured type prohibits the object itself from having values assigned to it.
- A READONLY actual VAR parameter can be passed only to a READONLY formal VAR parameter.
- A pointer expression whose base type is READONLY is assignment compatible only with a pointer variable whose base type is also READONLY.

Example

```
PROGRAM Test;

TYPE
   T = RECORD
       I : INTEGER;
       END;
   PRonly = ^ [READONLY] T;

VAR
   Pro : PRonly;
   Prw : ^ T;

PROCEDURE Q
   (P : PRonly);

   VAR
      X : INTEGER;

   BEGIN
       .
       .
       .
   X := P^.I;
       .
       .
       .
   END;

BEGIN
NEW (Pro);
NEW (Prw);
Q (Pro);
Q (Prw);
Prw^.I := 0;
    .
    .
    .
```

This example shows the declaration of two pointer variables, Pro and Prw, and the calls to NEW that create the dynamic variables Pro^ and Prw^. The type of the formal parameter P requires that a corresponding actual parameter have read access; therefore, both Pro and Prw can legally be passed to Q as actual parameters. Since P is a READONLY parameter, the value of the dynamic variable P^ (which corresponds to either Pro^ or Prw^) can be assigned to a variable, as shown in the assignment statement in the body of Q. However, only Prw^ can have values assigned to it, as shown in the last statement above.

10.17 Size Attributes

Size attributes can be applied to variables, formal parameters, base types of pointer variables, components of structured variables, and function results. They specify the amount of storage to be reserved for the object.

BIT[[(n)]]
BYTE[[(n)]]
WORD[[(n)]]
LONG[[(n)]]
QUAD[[(n)]]
OCTA[[(n)]]

The amount of storage may be expressed in bits, bytes, words, longwords, quadwords, or octawords. The optional constant n indicates the number of storage units.

Rules and Defaults

- The default size of an object depends on its type. See the *VAX-11 PASCAL User's Guide* for the rules of default allocation sizes.
- The constant expression n must denote a positive integer. If you omit n, the default value is 1.
- In VAX-11 PASCAL, the following size rules apply:
  - Objects of ordinal types cannot have sizes larger than 32 bits.
  - Objects of REAL, SINGLE, and pointer types must have sizes of exactly 32 bits.
  - Objects of type DOUBLE must have sizes of 64 bits.
  - Objects of type QUADRUPLE must have sizes of 128 bits.
- The amount of storage described must be large enough to contain an object of the specified type; otherwise, a compile-time error occurs.
- The size specified for an object of a structured type must be large enough to contain all the components of the object.
- A size attribute is illegal on a conformant parameter, a component of a VARYING string, and an object of a structured type having a file component. In an array variable passed to a conformant formal parameter, size and alignment attributes (see Section 10.2) are illegal on all dimensions of the actual parameter, except the first, that correspond to the dimensions of the formal parameter.
- Two variables of the same type that have different allocation sizes are assignment compatible, but are not structurally compatible.

Example

```
PROGRAM Size;

TYPE
   Status = [LONG] BOOLEAN;

VAR
   Return_Status : Status;

FUNCTION Example
   (Param1, Param2 : INTEGER)
   : Status;
   EXTERNAL;
      .
      .
      .
```

The program Size defines a Boolean type Status and declares a variable Return_Status of this type. Therefore, the result type of the function is declared to have a size of one longword.

10.18 UNBOUND Attribute

The UNBOUND attribute can be applied to routines and formal routine parameters. An UNBOUND routine does not access automatic variables in the scope in which it is declared. That is, the bound procedure value of an UNBOUND routine does not include the static scope pointer. The *VAX-11 PASCAL User's Guide* explains the use of the bound procedure value.

Rules and Defaults

- In the absence of an UNBOUND attribute, the compiler assumes that the bound procedure value of a routine includes the static scope pointer.
- By default, all predeclared routines and all routines declared at program or module level have the characteristics of UNBOUND routines. All routines declared in nested blocks are considered bound unless they have an UNBOUND attribute.
- All routines called from within the block of an UNBOUND routine must be local to the UNBOUND routine or be themselves UNBOUND, whether by default or by an explicit attribute.
- Nonlocal variables accessed from within the block of an UNBOUND routine cannot have AUTOMATIC allocation (see Section 10.3).
- If a formal routine parameter is UNBOUND, all actual routine parameters passed to it must also be UNBOUND.
- An UNBOUND routine may be passed as an actual parameter to a formal routine parameter that is not UNBOUND.

Example

```
[EXTERNAL] FUNCTION F
   (%IMMED [UNBOUND] PROCEDURE Count)
   : BOOLEAN;
   EXTERNAL;

PROCEDURE A;

   VAR
      I : [STATIC] INTEGER;
      B : BOOLEAN;

   [UNBOUND] PROCEDURE P;

      BEGIN
          .
          .
          .
      I := I + 1;
          .
          .
          .
      END;

   BEGIN
       .
       .
       .
   B := F(P);
       .
       .
       .
   END;
```

This example illustrates the declaration of the UNBOUND procedure P and the UNBOUND formal procedure parameter Count. Note that the executable section of P cannot access variables declared in the enclosing block of procedure A unless those variables are statically allocated. Thus, Handler can access the variable I, which is declared with the STATIC attribute, but cannot access the variable B, which is automatically allocated. Because the formal parameter Count is UNBOUND, only other UNBOUND routines (such as P) can be passed to function F as actual parameters. Count must be declared UNBOUND because it is passed by immediate value (see the *VAX-11 PASCAL User's Guide* for more information).

10.19 UNSAFE Attribute

The UNSAFE attribute can be applied to variables, formal parameters, the base types of pointer variables, components of structured variables, function results, and the types of other data items (see Table 10–2). UNSAFE indicates that an object can accept values of any type without type checking. The exact properties of an UNSAFE object depend on the object's machine representation.

Rules and Defaults

- A conformant VARYING parameter may not be declared UNSAFE.
- An expression of any type is assignment compatible with an UNSAFE object. However, neither the expression nor the object can contain a file component. If the machine representations of the expression and the UNSAFE object differ, the compiler forces them to have the same number of bits by modifying the value of the expression as follows:
  - If the expression contains more bits than the object, the low-order bits of the expression are assigned to the object and the high-order bits are discarded.
  - If the expression contains fewer bits than the object, the expression is assigned to the low-order bits of the object and the remaining high-order bits of the object are assigned zeros.
- A pointer expression is assignment compatible with a pointer variable whose base type is UNSAFE only if the base types have the same allocation size and if they have compatible alignment, READONLY, VOLATILE, and WRITEONLY attributes.
- An actual parameter variable can be passed to an UNSAFE formal VAR parameter if the types have the same allocation size and if they have compatible alignment, READONLY, VOLATILE, and WRITEONLY attributes.
- When a formal parameter is an UNSAFE conformant array, the VAX-11 PASCAL compiler must be able to establish bounds for the corresponding actual parameter that exactly describe the amount of storage the parameter occupies. If the conformant array is one-dimensional, the actual parameter need not be an array. The compiler constructs the bounds of the formal array so that the actual parameter and the formal array have the same size. For this construction to be possible, the size of the actual parameter must be an exact multiple of the size of the formal array component. The compiler chooses the low bound of the formal parameter's index to be the smallest possible nonnegative value of the index type. If the formal conformant parameter is a multidimensional array with n dimensions, the actual parameter must be an array having no fewer than n–1 dimensions. The first n–1 dimensions of the two arrays will have identical array bounds. The compiler chooses bounds for the last dimension of the conformant array so that the conformant as a whole describes the exact size of the actual parameter.

Example

```
PROGRAM Output_Buffer (DataFile);

TYPE
   Natural = 0..MAXINT;

VAR
   DataFile : FILE OF ARRAY[0..511] OF CHAR;
   Int_Array : ARRAY[0..1023] OF INTEGER;
   String : VARYING[2048] OF CHAR;
   Chr_Array : ARRAY[0..4095] OF CHAR;
   Status : BOOLEAN;

FUNCTION Put_Buf
   (VAR Buffer : [UNSAFE] ARRAY[A..B: Natural] OF CHAR)
   : BOOLEAN;

   VAR
      Cur : [STATIC] INTEGER := 0;
      I : INTEGER;

   BEGIN
   FOR I := A TO B DO
      BEGIN
      DataFile^[Cur] := Buffer[I];
      Cur := Cur + 1;
      IF Cur > 511
      THEN
         BEGIN
         PUT(DataFile);
         Cur := 0;
         END;
      END;
   Put_Buf := (Cur = 0);
   END;

BEGIN (* Main program *)
   .
   .
   .
Status := Put_Buf (Int_Array);
Status := Put_Buf (String);
Status := Put_Buf (Chr_Array);
   .
   .
   .
END.
```

The program Output_Buffer declares a function whose only formal parameter is an UNSAFE conformant array of characters. The function Put_Buf assigns successive components of the conformant array parameter to the file buffer variable of DataFile. If DataFile^ is filled, the function returns TRUE; otherwise, it returns FALSE.

The program issues three calls to Put_Buf. In the first and second calls, the actual parameters are not of the same type as the formal parameter Buffer. But, because Buffer has the UNSAFE attribute, it accepts an actual parameter of any type and treats it as though it were an array of characters. The third call to Put_Buf passes an actual parameter of the same type as the formal parameter.

10.20 Visibility Attributes

The visibility attributes can be applied to variables, routines, and compilation units. They control the sharing of an object between independently compiled units and indicate the name by which the object is known outside the compilation unit that declares it. See the *VAX-11 Linker Reference Manual* for further information.

LOCAL

The LOCAL attribute indicates that an object is unavailable to other independently compiled units. Only compilation units that have access to the environment in which the object was declared can refer to the object.

GLOBAL [[(identifier)]]

The GLOBAL attribute provides a strong definition of an object so that other independently compiled units can refer to it. You can specify an identifier with the GLOBAL attribute to indicate the name by which the corresponding object is known to the VAX-11 Linker. Normally, you do not include an identifier, so that the linker recognizes the same object name as the declaring compilation unit.

EXTERNAL [[(identifier)]]

The EXTERNAL attribute indicates a variable or routine that is assumed to be GLOBAL in another independently compiled unit. If the attribute includes an identifier, that name, rather than the identifier being declared, is supplied to the VAX-11 Linker. The names available to the linker for corresponding GLOBAL and EXTERNAL variables and routines must be identical.

WEAK__GLOBAL [[(identifier)]]
WEAK__EXTERNAL [[(identifier)]]

These attributes are similar to the GLOBAL and EXTERNAL attributes. A WEAK__GLOBAL object is linked only when it is specifically included in the linking operation. A WEAK__EXTERNAL variable or routine is not critical to the linking operation. To resolve a weak reference, the linker searches only the named input modules.

Rules and Defaults

- By default, all variables and routines are LOCAL.
- Compilation units may not have the EXTERNAL or WEAK__EXTERNAL attribute.
- Variables with any visibility attribute other than LOCAL are implicitly static.
- LOCAL is the only visibility attribute you can specify for nonstatic variables.
- By default, GLOBAL and EXTERNAL routines have the characteristics of UNBOUND routines (see Section 10.18).

- Routines with any visibility attribute other than LOCAL cannot refer to AUTOMATIC variables declared in enclosing blocks (see Section 10.3) and can call only those routines that are local, predeclared, or unbound (by default, routines declared at program or module level have the characteristics of UNBOUND routines).
- EXTERNAL routines must be followed by the directive EXTERN, EXTERNAL, or FORTRAN when they are declared (see Section 6.5.2).

Example

```
PROGRAM Freshman_Class;
[GLOBAL(Sort_Students)] PROCEDURE Class_List
   (VAR Register_List;
   Sorted_List : Student_Rec);

MODULE Senior_Class;
[EXTERNAL(Sort_Students)] PROCEDURE Roll_Call
   (VAR Start_List;
   End_List : Senior_Rec);
```

This example shows the global declaration of a procedure with the name Sort_Students and an external reference to the same procedure in a different compilation unit.

10.21 VOLATILE Attribute

The VOLATILE attribute can be applied to variables, formal parameters, the base types of pointer variables, components of structured variables, and function results. VOLATILE indicates the assumptions that the compiler can legally make about the value of an object. Normally, a compiler assumes that an object's value will not be subject to unusual side effects. During execution, an object's value will generally change only under the following circumstances:

- When another value is assigned to it
- When it is passed as a writeable VAR parameter
- When it is read into by a READ, READLN, or READV procedure
- When it is used as the control variable of a FOR loop

In addition, the compiler expects to evaluate the object only when it appears in an expression.

The VOLATILE attribute informs the compiler that the object's value will be subject to unusual side effects during execution. In addition to changing in the usual ways, the value of a VOLATILE object may change as the result of an action not directly specified in the program. Thus, the compiler assumes that the value of a VOLATILE object can be changed or evaluated at any time during program execution. Consequently, a VOLATILE object does not participate in any optimization based on assumptions about its value.

Examples of VOLATILE behavior are the behavior of many device registers and modification by asynchronous processes and exception handlers.

Rules and Defaults

- By default, objects are not VOLATILE.
- An object of a structured type that has a VOLATILE component is VOLATILE as a whole. However, the presence of a VOLATILE component does not make other components of the same variable VOLATILE.
- The presence of the VOLATILE attribute guarantees that operations will be performed on scalar objects in a single machine instruction. Because operations on structured objects may require more than one instruction, the use of the VOLATILE attribute on an object of a structured type might not produce the expected results.
- A VOLATILE variable is structurally compatible only with a formal VAR parameter that is VOLATILE.
- A pointer expression whose base type is VOLATILE is assignment compatible only with a pointer variable whose base type is VOLATILE.
- Two pointer types are structurally compatible only if their base types have identical volatility.

Examples

1.
```
VAR
    X : CHAR;
    A : [VOLATILE] RECORD
        CASE BOOLEAN OF
            FALSE : (I : INTEGER);
            TRUE : (C : CHAR);
        END;
    .
    .
    .
A.C := 'A'; (* TRUE becomes the current variant *)
A.I := 66;  (* Assignment makes FALSE the current variant *)
X := A.C;   (* TRUE is again the current variant;
               X is assigned the value 'B', which
               has an ordinal value of 66 *)
    .
    .
    .
```

As the comments show, a reference to one field identifier causes the corresponding variant to become the current variant. In addition, each reference immediately causes the other variant to become undefined. Thus, when the assignment A.I := 66 is made, the reference to A.I causes FALSE to become the current variant and A.C to become undefined. As a result of the statement X := A.C, the value last assigned to the variant is assigned to X. Ordinarily the compiler could assume that A.C had retained the value 'A', since no further assignments had been made directly to A.C. However, the value of A.C changed unexpectedly through the assignment to A.I. Therefore, unless the record A is declared VOLATILE, the result of the assignment X := A.C would be undefined because the compiler's legitimate assumptions had been violated.

2.
```
PROGRAM Volatility (OUTPUT);

VAR
   Pint : ^[VOLATILE] INTEGER;
   I : INTEGER;
   J : [VOLATILE] INTEGER;
   A : ARRAY[0..10] OF INTEGER;

BEGIN
NEW (Pint);
I := 0;
J := 0;
Pint^ := 0;

(* Compiler may assume I = 0, makes no assumptions about J *)

WRITELN (I, J, Pint^, A[I]);  (* Values are 0, 0, 0, A[0] *)
Pint := ADDRESS (J);          (* Pint^ now = J *)
Pint^ := 1;                   (* Therefore J now = 1 *)

(* Compiler may assume I = 0, makes no assumptions about J *)

WRITELN (I, J, Pint^, A[I]);  (* Values are 0, 1, 1, A[0] *)
Pint := ADDRESS (I);          (* Causes a warning message
                                 since I is not VOLATILE *)
Pint^ := 2;

(* Compiler may assume I = 0 and A[I] = A[0]
   May make no assumptions about J *)

WRITELN (I, J, Pint^, A[I]);  (* Actual values are
                                 2, 1, 2, A[2] *);
END.
```

This example assigns values to the variables I and J and to the newly created variable Pint^. The comments illustrate the difference between the assumptions the compiler can legally make about the values of the variables and the actual values contained in the variables. The compiler's assumption about the value of I was incorrect because the value of I changed unexpectedly. The ADDRESS (I) function caused Pint to point to I (that is, Pint^ and = I became the same variable). When Pint^ was assigned the value 2, I also received the value 2. Because I had been initialized to 0 and was not directly referred to in the rest of the program, the compiler assumed that a reference to I at this point would be equivalent to a reference to 0. Likewise, the compiler also assumed that a reference to A[I] would be equivalent to a reference to A[0]. In fact, however, when execution ceases, the value of I is 2 and the value of A[I] is the value of A[2].

Depending on the optimizations the compiler made about the value of I, any operations performed after the unanticipated assignment to I could yield unexpected results. Because J was declared VOLATILE, the compiler did not optimize code based on the value of J. Therefore, any reference to J yields the expected results.

Note that the ADDRESS (I) function in this program causes a warning message. The VAX-11 PASCAL compiler assumes that pointer variables point only to variables in heap-allocated storage and not to statically allocated, nonvolatile variables such as I. Thus, ADDRESS (I) in this case violates the compiler's assumptions.

10.22 WRITEONLY Attribute

The WRITEONLY attribute can be applied to variables, formal parameters, the base types of pointer variables, and components of structured variables. WRITEONLY specifies that an object can have values assigned to it but cannot be read by a program. For example, if X is a WRITEONLY integer variable, you can give it a new value by assignment, as in X := 23, or by reading a new value into it, as in READ (X). But you cannot assign the value of X to another variable, as in Y := X.

Rules and Defaults

- By default, objects can be both read and written.
- A WRITEONLY object cannot be used in expressions.
- A WRITEONLY component in an object of a structured type prohibits the object itself from being written.
- A WRITEONLY actual VAR parameter can be passed only to a formal VAR parameter that is WRITEONLY.
- A pointer expression whose base type is WRITEONLY is assignment compatible only with a pointer variable whose base type is WRITEONLY.

Example

```
TYPE
   WOnly = [WRITEONLY] INTEGER;

VAR
   Writ_Int : WOnly;
   Norm_Int : INTEGER;

PROCEDURE Try_Access
   (VAR Write_Param : WOnly);
   .
   .
   .
BEGIN (* Main program *)
Writ_Int := SQR (Norm_Int);
Try_Access (Writ_Int);
   .
   .
   .
```

This example shows legal statements involving WRITEONLY variables. The WRITEONLY variable Writ_Int is assigned the result of the square root operation and is passed as an actual parameter to a WRITEONLY formal parameter.

Appendix A
ASCII Character Set

Table A–1 summarizes the ASCII character set. Each element of the character set is a constant of the predefined PASCAL type CHAR. An 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.

Note that VAX–11 PASCAL uses an extended implementation of the ASCII character set. The extended characters, which do not appear in Table A–1, have the following decimal values:

- 128–160 Extended control characters
- 161–254 Extended graphics characters
- 255 "Eight Ones"

Table A–1: The ASCII Character Set

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
0	NUL	Null	21	NAK	Negative acknowledgement
1	SOH	Start of heading	22	SYN	Synchronous idle
2	STX	Start of text	23	ETB	End of transmission block
3	ETX	End of text	24	CAN	Cancel
4	EOT	End of transmission	25	EM	End of medium
5	ENQ	Enquiry	26	SUB	Substitute
6	ACK	Acknowledgement	27	ESC	Escape
7	BEL	Bell	28	FS	File separator
8	BS	Backspace	29	GS	Group separator
9	HT	Horizontal tab	30	RS	Record separator
10	LF	Line feed	31	US	Unit separator
11	VT	Vertical tab	32	SP	Space or blank
12	FF	Form feed	33	!	Exclamation mark
13	CR	Carriage return	34	"	Quotation mark
14	SO	Shift out	35	#	Number sign
15	SI	Shift in	36	$	Dollar sign
16	DLE	Data link escape	37	%	Percent sign
17	DC1	Device control 1	38	&	Ampersand
18	DC2	Device control 2	39	'	Apostrophe
19	DC3	Device control 3	40	(	Left parenthesis
20	DC4	Device control 4	41	)	Right parenthesis

Table A-1 (Cont.): The ASCII Character Set

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
42	*	Asterisk	85	U	Uppercase U
43	+	Plus sign	86	V	Uppercase V
44	,	Comma	87	W	Uppercase W
45	-	Minus sign or hyphen	88	X	Uppercase X
46	.	Period or decimal point	89	Y	Uppercase Y
47	/	Slash	90	Z	Uppercase Z
48	0	Zero	91	[	Left square bracket
49	1	One	92	\	Back slash
50	2	Two	93	]	Right square bracket
51	3	Three	94	^ or ↑	Circumflex or up arrow
52	4	Four	95	← or _	Back arrow or underscore
53	5	Five	96	`	Grave accent
54	6	Six	97	a	Lowercase a
55	7	Seven	98	b	Lowercase b
56	8	Eight	99	c	Lowercase c
57	9	Nine	100	d	Lowercase d
58	:	Colon	101	e	Lowercase e
59	;	Semicolon	102	f	Lowercase f
60	<	Left angle bracket	103	g	Lowercase g
61	=	Equal sign	104	h	Lowercase h
62	>	Right angle bracket	105	i	Lowercase i
63	?	Question mark	106	j	Lowercase j
64	@	At sign	107	k	Lowercase k
65	A	Uppercase A	108	l	Lowercase l
66	B	Uppercase B	109	m	Lowercase m
67	C	Uppercase C	110	n	Lowercase n
68	D	Uppercase D	111	o	Lowercase o
69	E	Uppercase E	112	p	Lowercase p
70	F	Uppercase F	113	q	Lowercase q
71	G	Uppercase G	114	r	Lowercase r
72	H	Uppercase H	115	s	Lowercase s
73	I	Uppercase I	116	t	Lowercase t
74	J	Uppercase J	117	u	Lowercase u
75	K	Uppercase K	118	v	Lowercase v
76	L	Uppercase L	119	w	Lowercase w
77	M	Uppercase M	120	x	Lowercase x
78	N	Uppercase N	121	y	Lowercase y
79	O	Uppercase O	122	z	Lowercase z
80	P	Uppercase P	123	{	Left brace
81	Q	Uppercase Q	124	|	Vertical line
82	R	Uppercase R	125	}	Right brace
83	S	Uppercase S	126	~	Tilde
84	T	Uppercase T	127	DEL	Delete

Appendix B
Syntax Summary

This appendix summarizes the syntax of VAX-11 PASCAL in the notation used throughout this manual and presents syntax diagrams in the format commonly used for PASCAL.

B.1 Syntax Productions

This section provides the collected syntax productions that define VAX-11 PASCAL.

Syntax

actual-parameter-list ->

```
                  [[mechanism-specifier]]  procedure-identifier
                  [[mechanism-specifier]]  function-identifier
( {               [[mechanism-specifier]]  expression                    } ,...)
                  [[type-identifier    ,]] { constant-expression },...
                  write-list-element
```

array-constructor ->

({[[constant-expression OF]] initial-value },...)

array-type ->

[[PACKED]] ARRAY [{[[attribute-list]] simple-type },...] OF type

assignment-statement ->

```
{ variable          } := expression
{ function-identifier }
```

attribute-list ->

[{ identifier [[({ expression },...)]] },...]

binary-digits ->

```
{0} ...
{1}
```

```
block ->

   declaration-part
   BEGIN
   { statement };...
   END

case-statement ->

   CASE expression OF
        {{ constant-expression },... : statement };...
        [[;]] OTHERWISE { statement };...[]
        [;]
   END

compilation unit ->

      { program }
      { module  }

compound-statement->

   BEGIN
   { statement };...
   END

conformant-schema ->

{  VARYING [identifier] OF [[attribute-list]] type-identifier                    }
{                                                                                }
{  PACKED ARRAY [identifier..identifier : [[attribute-list]] type-identifier]    }
{            OF [[attribute-list]] type-identifier                               }
{                                                                                }
{  ARRAY [{ identifier..identifier : [[attribute-list]] type-identifier };...]   }
{            OF [[attribute-list]] { type-identifier    }                        }
{                                  { conformant-schema  }                        }

constant-definition ->

   CONST { identifier = constant-expression };...

decimal-digits ->

      { 0  1  2  3  4 }
      { 5  6  7  8  9 }  ...

declaration-part ->

{  label-declaration      }
{  constant-definition    }
{  type-definition        }
{  variable-declaration   }  ...
{  value-declaration      }
{  procedure-declaration  }
{  function-declaration   }
```

directive ->

{ EXTERN
  EXTERNAL
  FORTRAN
  FORWARD }

empty-statement ->

enumerated-type ->

({identifier},...)

expression ->

simple-expression [[{ <> < <= } simple-expression]]
                   { =   > >= IN }

extended-alphabetic ->

{ letter }
{ _   }
{ $ }

factor ->

{ array-type-identifier array-constructor
  constant-identifier
  (expression)[:: type-identifier]
  function-identifier [actual-parameter-list]
  NOT factor
  numeric-constant
  real-constant
  record-type-identifier [record-constructor]
  [set-type-identifier] set-constructor
  string-constant
  variable }

field-list ->

[[{ {identifier},... : type};... [; variant-clause] } [;]]]
   { variant-clause   }

file-type ->

[PACKED] FILE OF type

for-statement ->

FOR variable-identifier := expression { TO   } expression DO
                                      { DOWNTO }
  statement

foreign-section ->

mechanism-specifier { value-section | procedure-section | function-section }

formal-parameter-list ->

({ value-section | VAR-section | procedure-section | function-section | foreign-section } ;...)

function-declaration ->

function-heading ; { block | directive } ;

function-heading ->

[[attribute-list]] FUNCTION identifier [[formal-parameter-list]]
: [[attribute-list]] type-identifier

function-section ->

function-heading [[:= [[mechanism-specifier]] initial-value]]

goto-statement ->

GOTO decimal-digits

hexadecimal-digits ->

{ decimal-digits | A B C D E F | a b c d e f } ...

identifier ->

extended-alphabetic [[{ decimal-digits | extended-alphabetic } ...]]

if-statement ->

IF expression
THEN statement
[[ELSE statement]]

initial-value ->

{ constant-expression | array-constructor | record-constructor }

label-declaration ->

LABEL {decimal-digits},...;

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 }

mechanism-specifier ->

{ %DESCR
  %IMMED
  %REF
  %STDESCR }

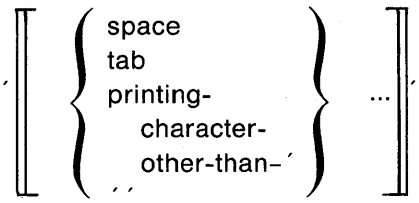
module ->

module-heading
declaration-part
END.

module-heading ->

[[attribute-list]] MODULE identifier [[({identifier},...)]];

name-string ->



numeric-constant ->

{ decimal-digits
  %O 'octal-digits'
  %X 'hexadecimal-digits'
  %B 'binary-digits' }

octal-digits ->

{ 0 1 2 3
  4 5 6 7 } ...

pointer-type ->

^ [[attribute-list]] type-identifier

primary ->

factor [{ ** factor }...]

procedure-declaration ->

procedure-heading ; { block / directive } ;

procedure-heading ->

[attribute-list] PROCEDURE identifier [formal-parameter-list]

procedure-section ->

procedure-heading [:= [mechanism-specifier] initial-value]

procedure-statement ->

{ procedure-identifier / function-identifier } [actual-parameter-list]

program ->

program-heading
block.

program-heading ->

[attribute-list] PROGRAM identifier [({ identifier },...)];

real-constant ->

decimal-digits { . decimal-digits / . decimal-digits { Q / D / E } { + / - } decimal-digits }

record-constructor ->

({initial-value},...)

record-type ->

[PACKED] RECORD
field-list
END

repeat-statement ->

REPEAT {statement};...UNTIL expression

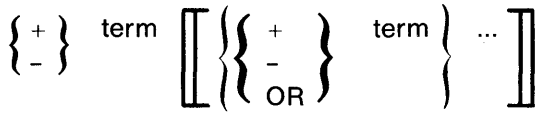
set-constructor ->

[[{ expression [..expression] } ,...]]

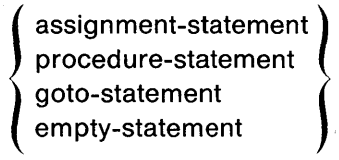
set-type ->

[[PACKED]] SET OF [[attribute-list]] simple-type

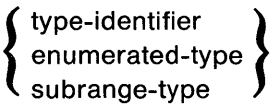
simple-expression ->



simple-statement ->



simple-type ->



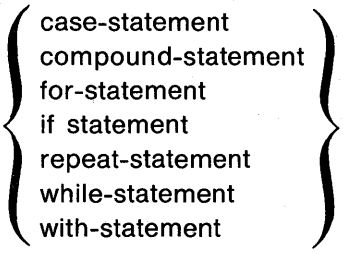
statement ->

[[decimal-digits :]] { simple-statement | structured-statement }

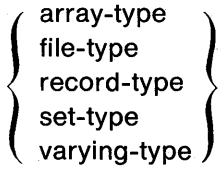
string-constant ->

{ name-string | {name-string ({constant-expression},...)}... [[name-string]] }

structured-statement ->



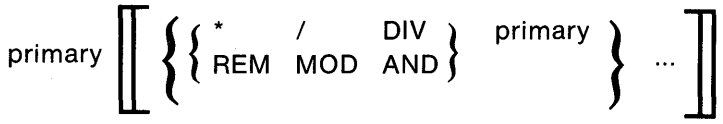
structured-type ->



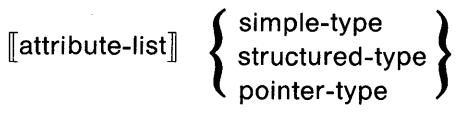
subrange-type ->

constant-expression..constant-expression

term ->



type ->



type-definition ->

TYPE {identifier = type};...

value-declaration ->

VALUE {identifier := initial-value};...

value-section ->

{identifier},... : [[attribute-list]] { type-identifier / conformant-schema }

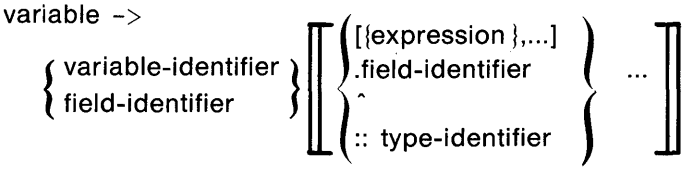
[[:= [[mechanism-specifier]] initial-value]]

VAR-section ->

VAR {identifier},... : [[attribute-list]] { type-identifier / conformant-schema }

[[:= [[mechanism-specifier]] initial-value]]

variable ->



variable-declaration ->

VAR {{identifier},... : type [[:= initial-value]]};...

variant-clause ->

CASE [[identifier :]] [[attribute-list]] type-identifier OF
{{constant-expression},... : (field-list)};...

varying-type ->

VARYING [constant-expression] OF [[attribute-list]] type-identifier

```
while-statement ->
    WHILE expression DO statement

with-statement ->
    WITH {variable},... DO  statement

write-list-element ->
    expression[[:expression[[:expression]]]]
```

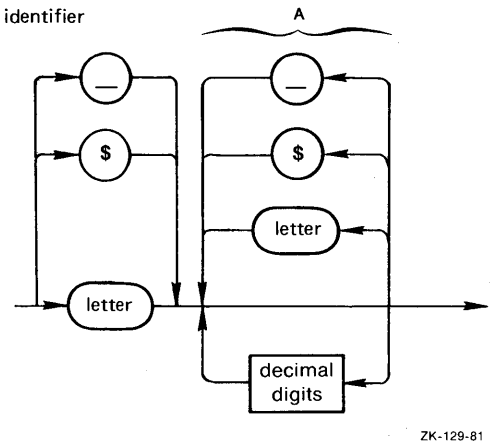
B.2 Syntax Diagrams

The following diagrams illustrate the syntax of these items:

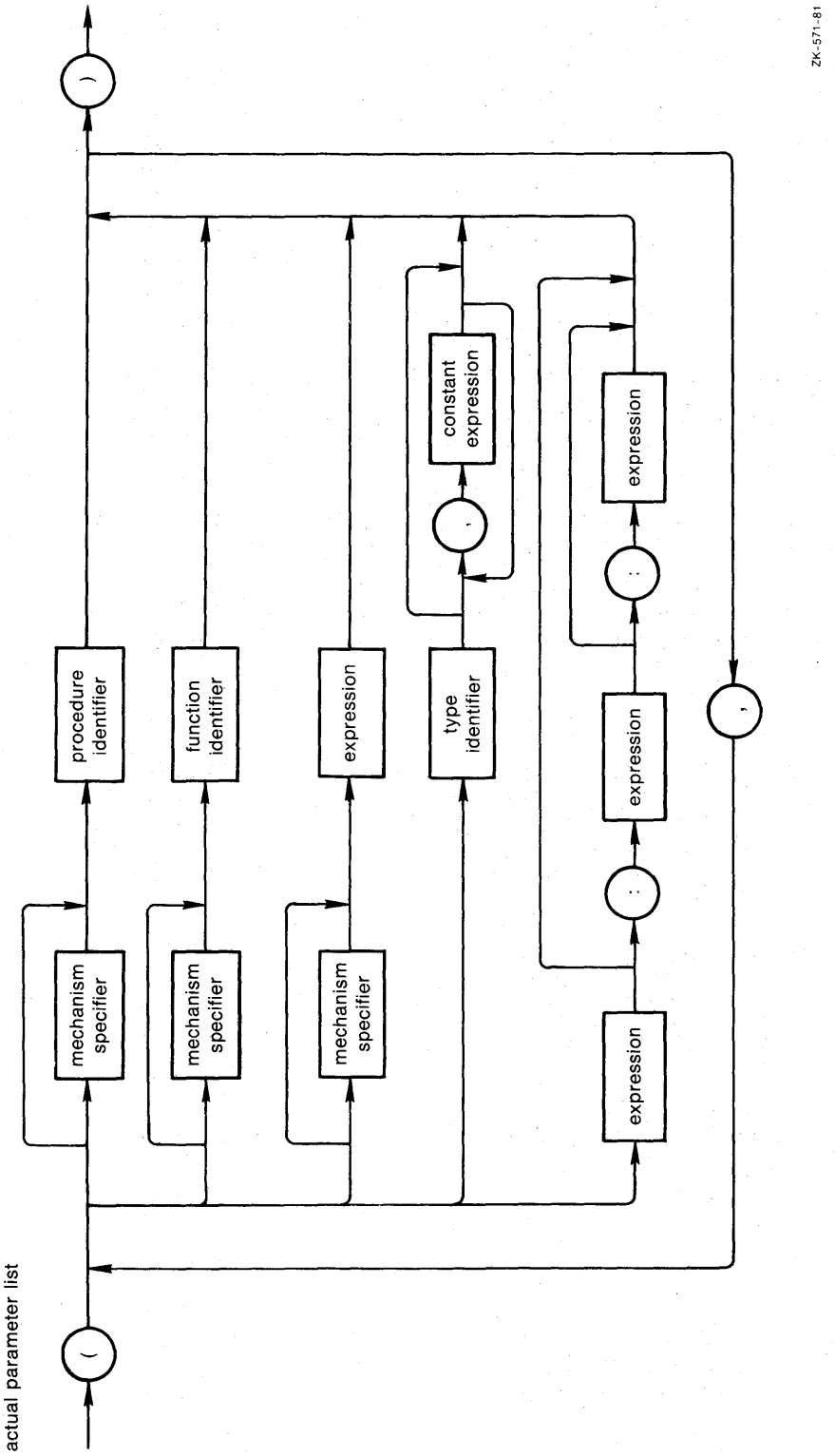
- Actual parameter list
- Array constructor
- Attribute list
- Binary digits
- Block
- Compilation unit
- Conformant schema
- Decimal digits
- Declaration part
- Expression
- Factor
- Field list
- Formal parameter list
- Formal parameter section
- Function heading
- Hexadecimal digits
- Identifier
- Initial value
- Mechanism specifier
- Numeric constant
- Octal digits
- Primary
- Procedure heading
- Real constant

- Record constructor
- Routine declaration
- Set constructor
- Simple expression
- Simple statement
- Simple type
- Statement
- String constant
- Structured statement
- Term
- Type
- Variable

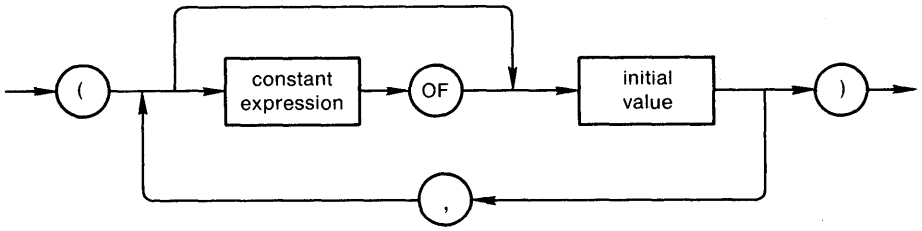
An example of how to interpret a diagram follows:



The diagram illustrates that the first character of an identifier can be either an underscore (_), a dollar sign ($), or a letter. The next character is chosen from the section labeled A and can be a digit, a letter, a dollar sign, or an underscore. Section A is repeated until the identifier is defined. Note that rounded symbols (circles or ovals) denote elements that must appear exactly as shown; rectangular symbols denote elements for which there is a separate diagram.

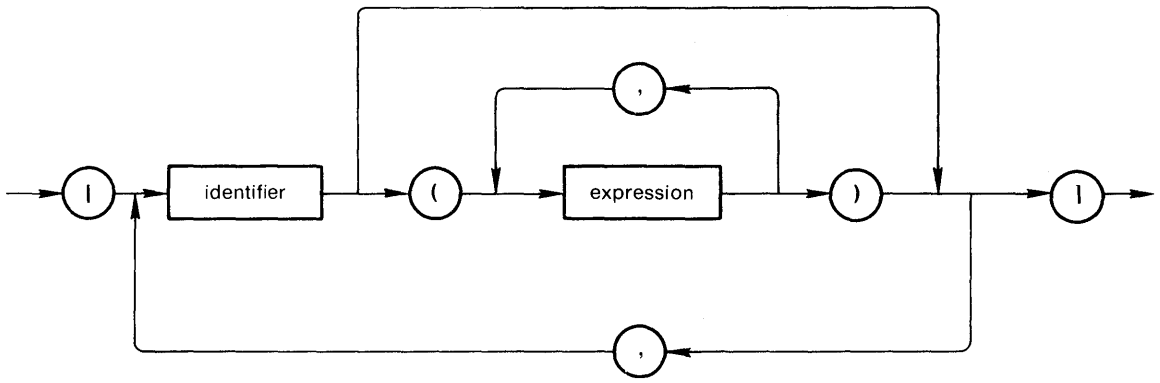

actual parameter list
(
)
mechanism specifier
procedure identifier
mechanism specifier
function identifier
mechanism specifier
expression
type identifier
constant expression
expression
expression
expression
ZK-571-81

array constructor



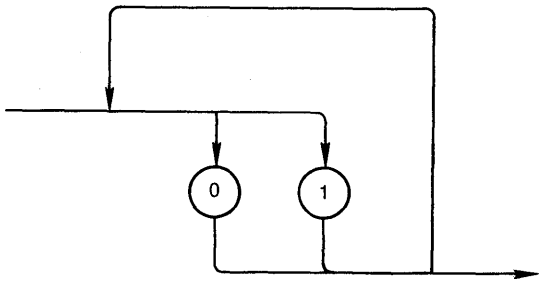
ZK-570-81

attribute list



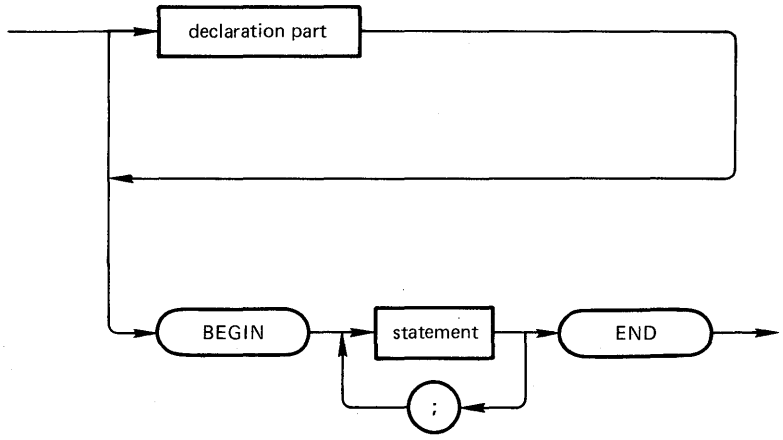
ZK-131-81

binary digits

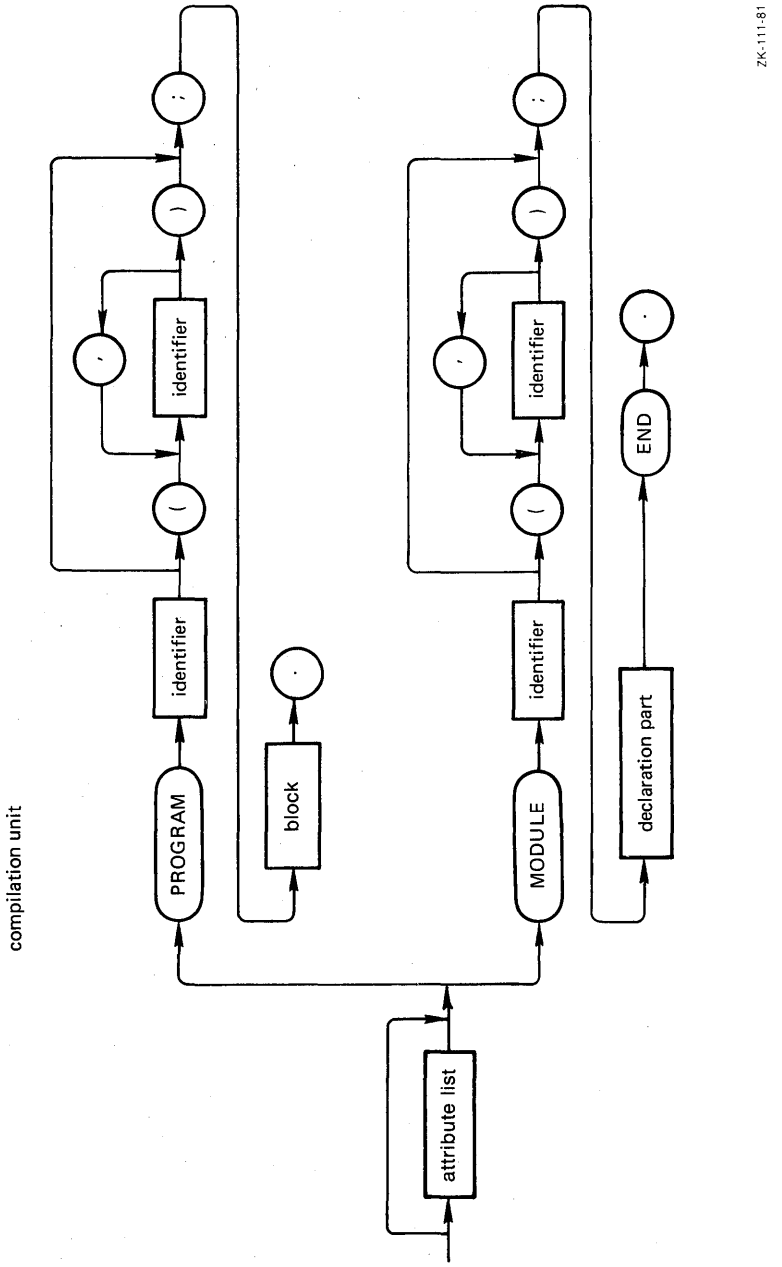


ZK-132-81

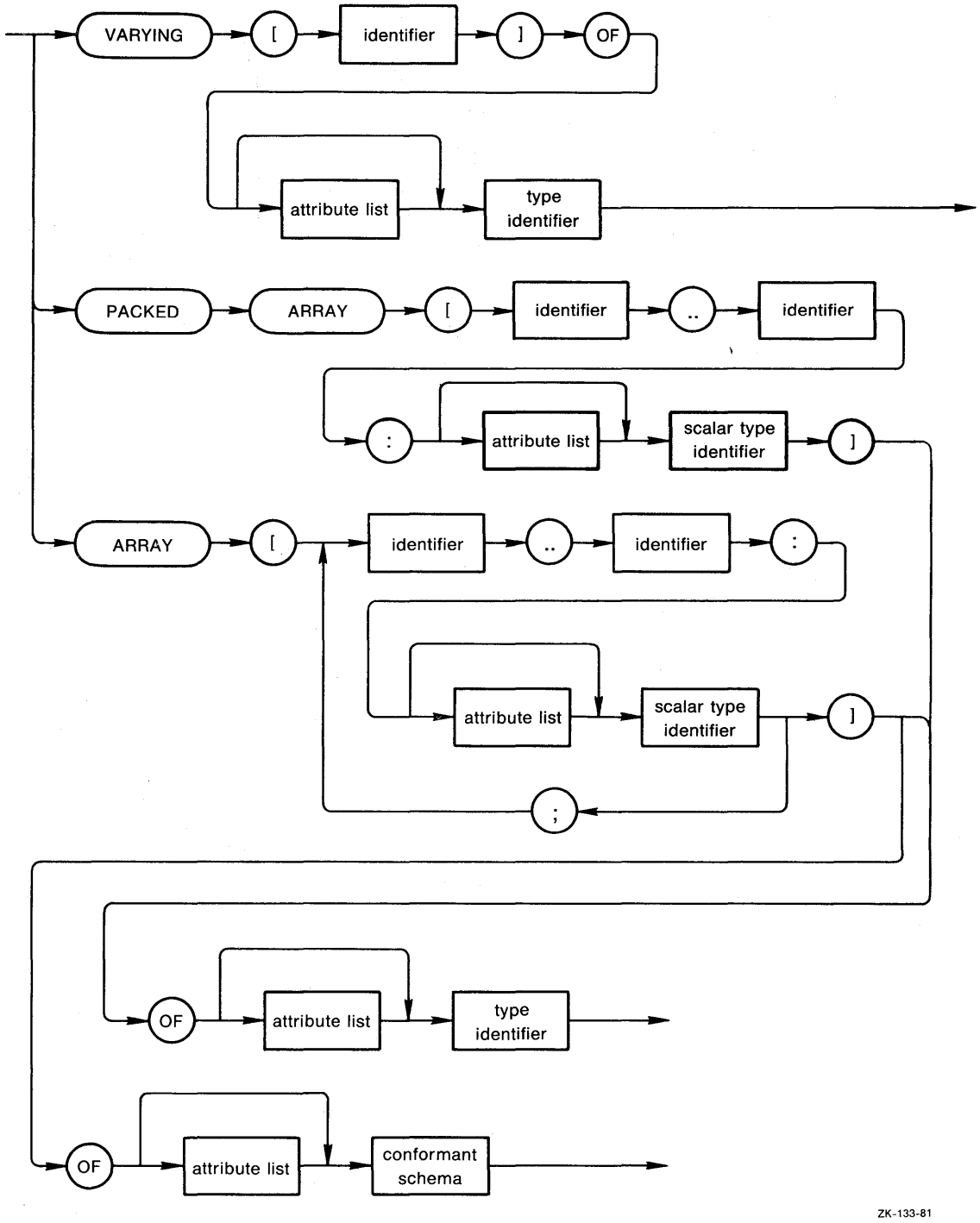
block



ZK-107-81

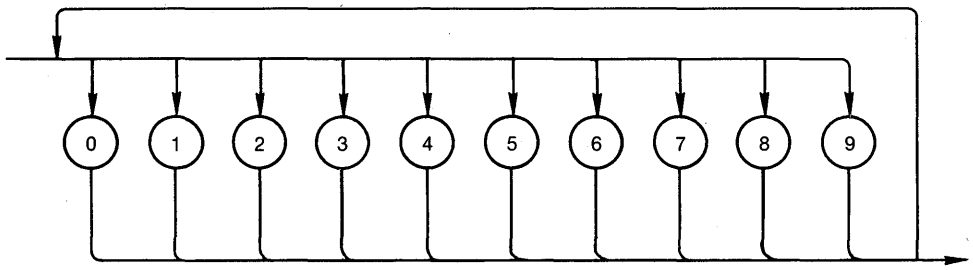

compilation unit
attribute list
PROGRAM
identifier
(
,
identifier
)
;
block
.
MODULE
identifier
(
,
identifier
)
;
declaration part
END
.
ZK-111-81

conformant schema

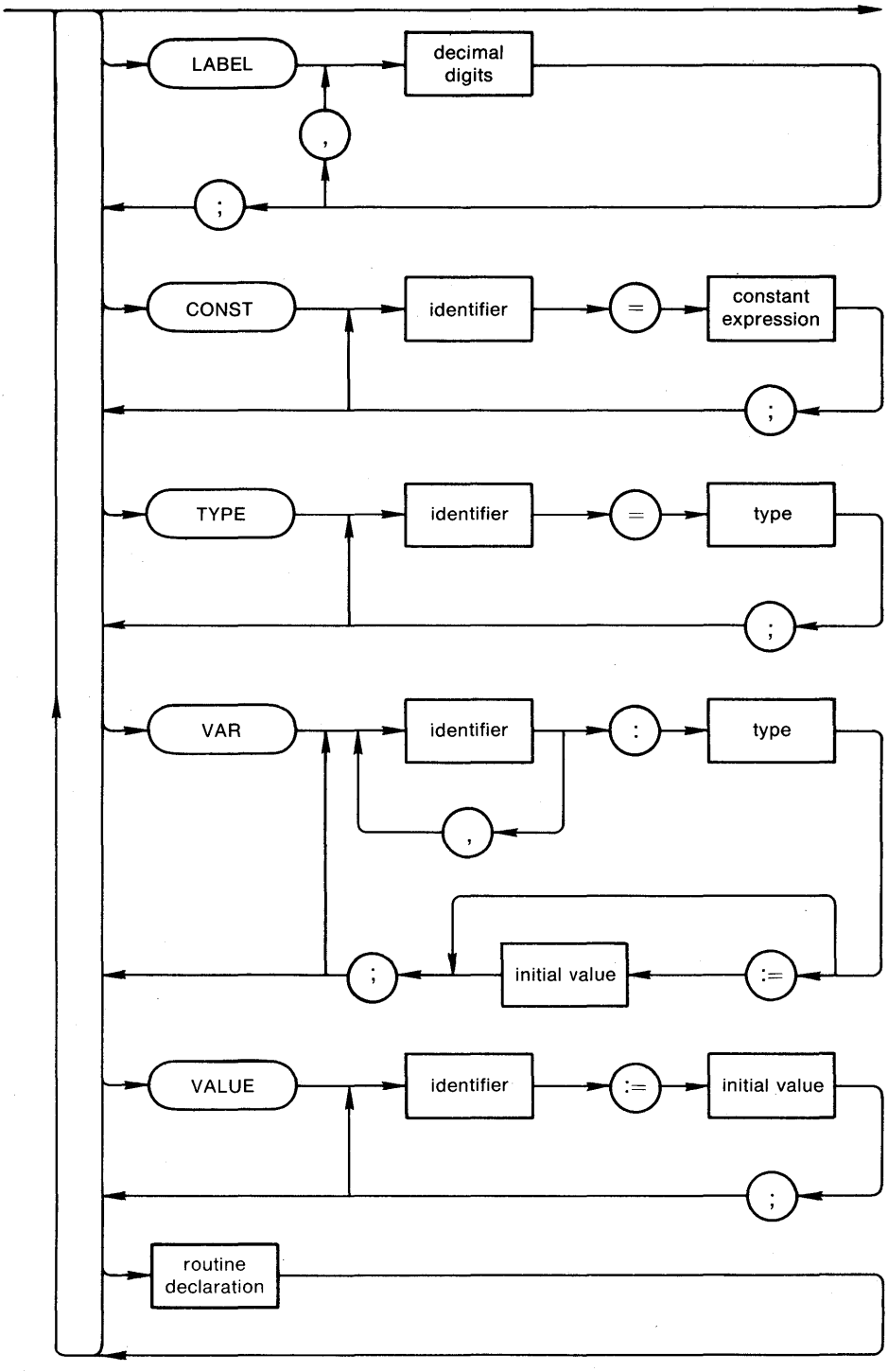


ZK-133-81

decimal digits

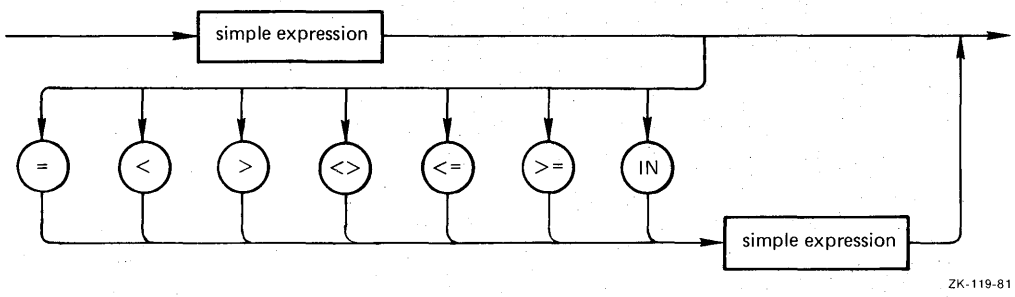


declaration part

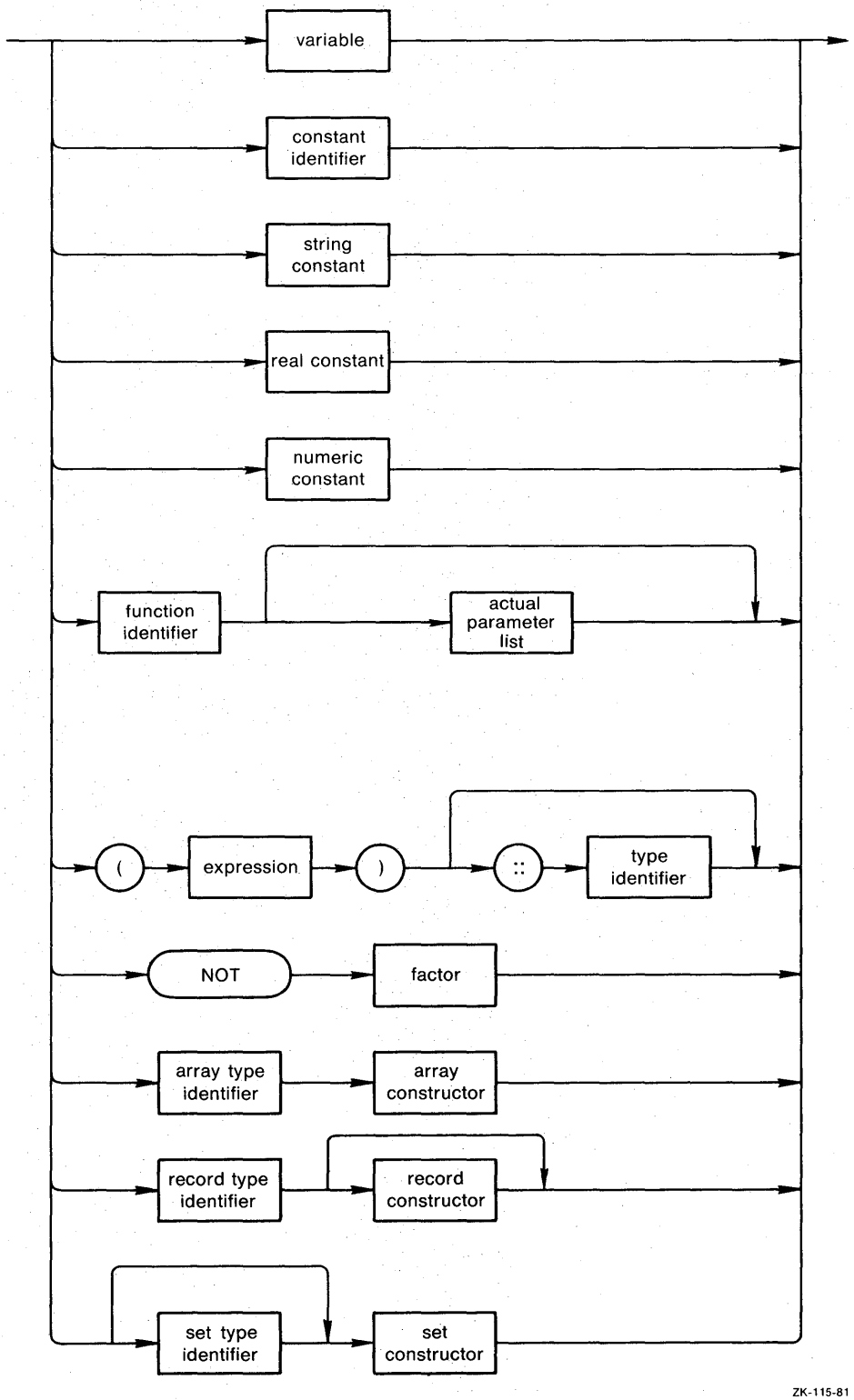


ZK-109-81

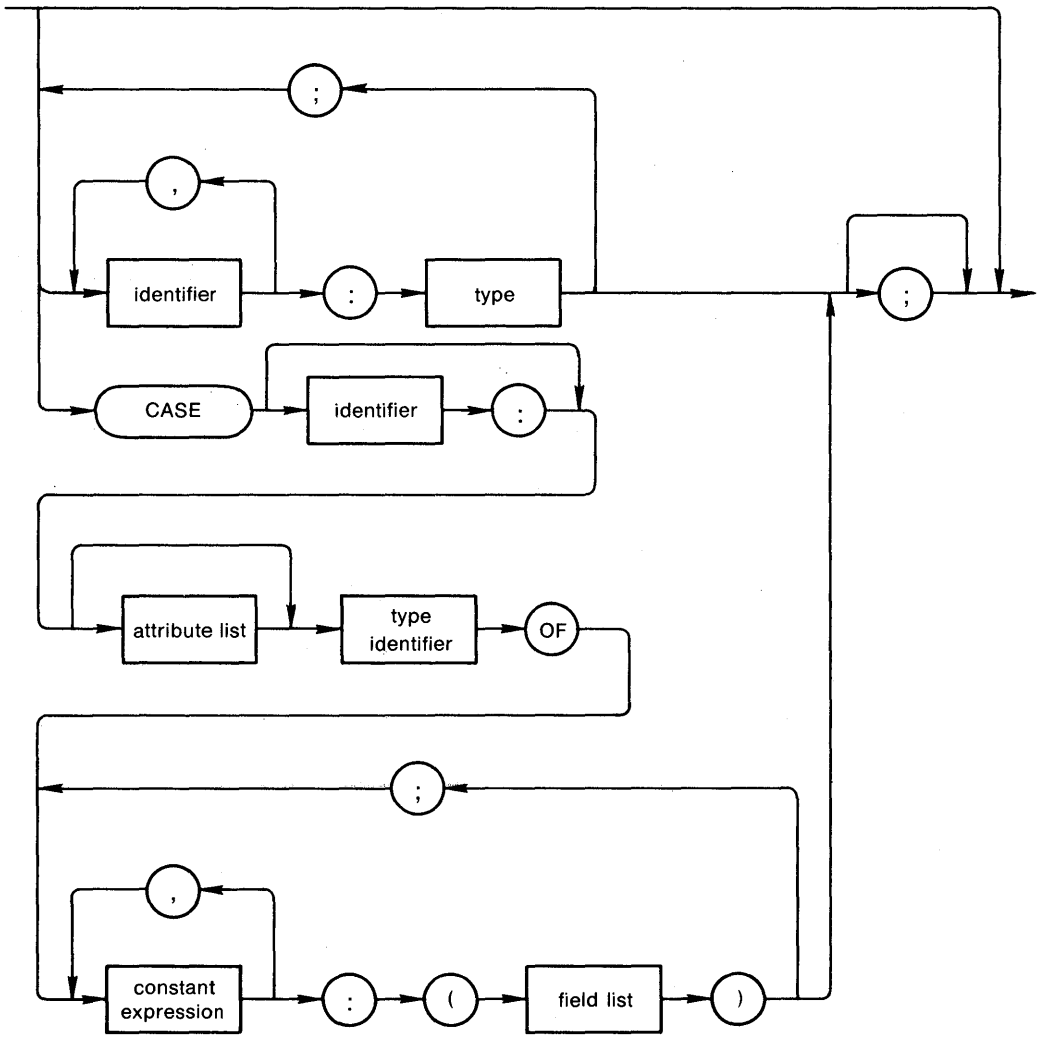
expression



factor

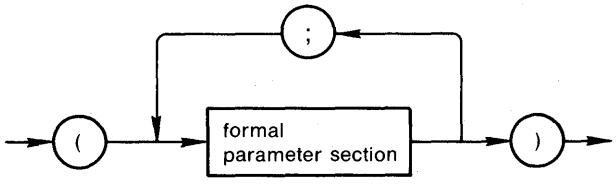

variable
constant identifier
string constant
real constant
numeric constant
function identifier
actual parameter list
(
expression
)
::
type identifier
NOT
factor
array type identifier
array constructor
record type identifier
record constructor
set type identifier
set constructor
ZK-115-81

field list



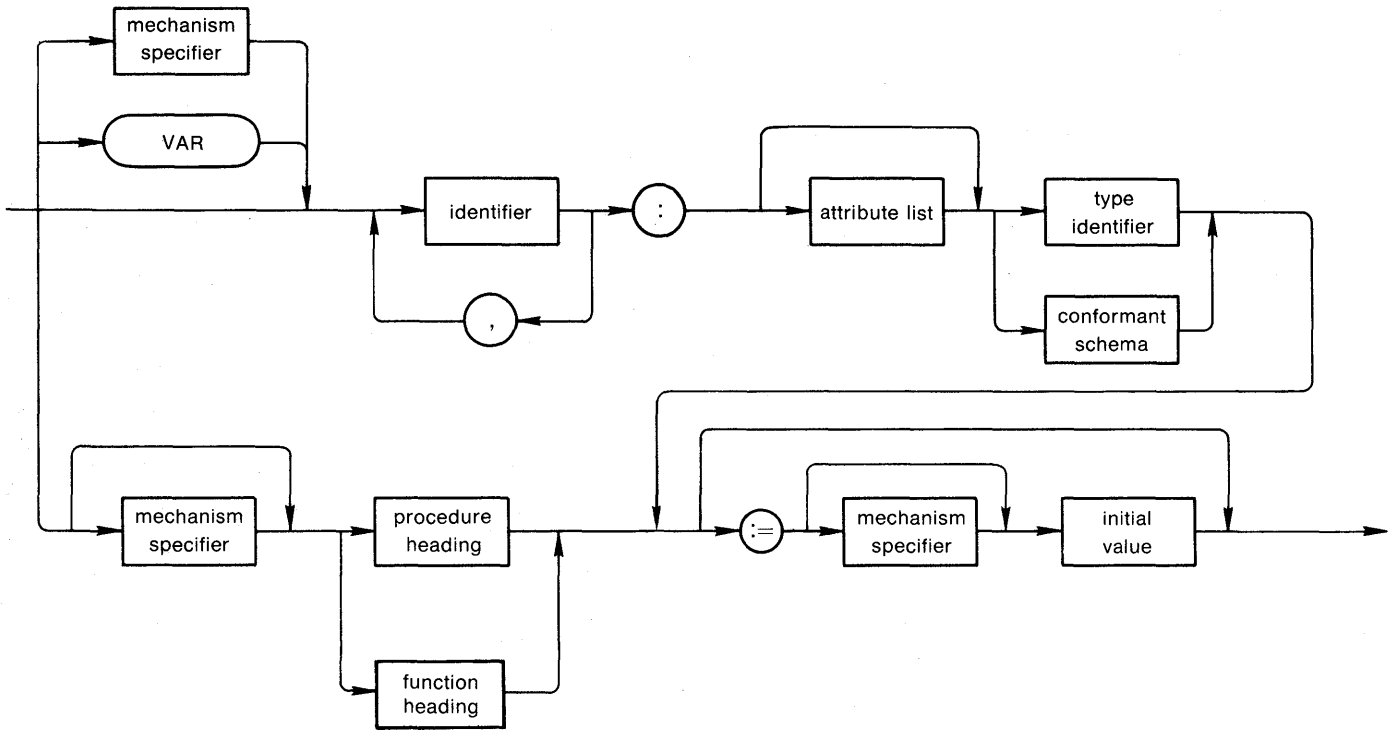
ZK-134-81

formal parameter list



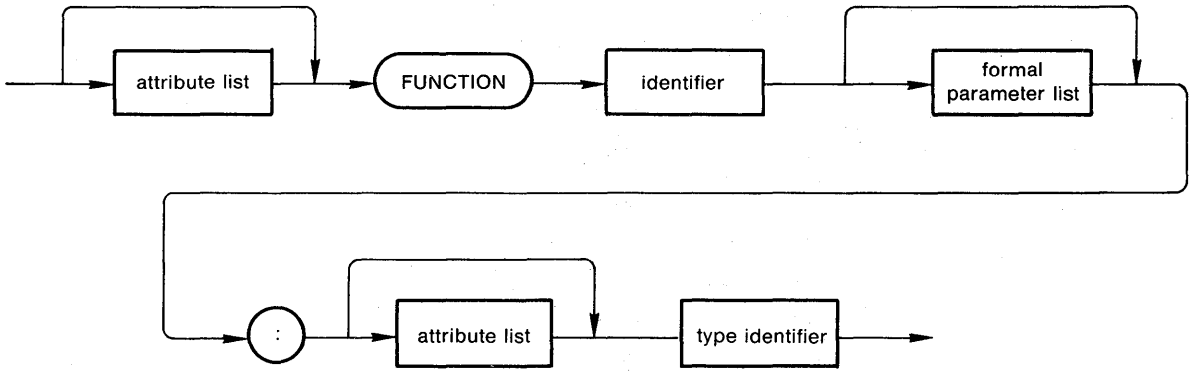
ZK-568-81

formal parameter section



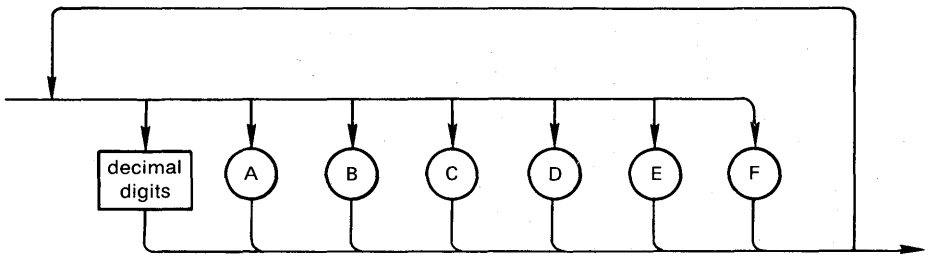
ZK-136-81

function heading



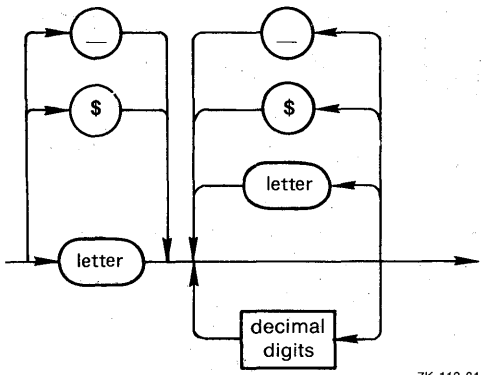
ZK-1035-82

hexadecimal digits



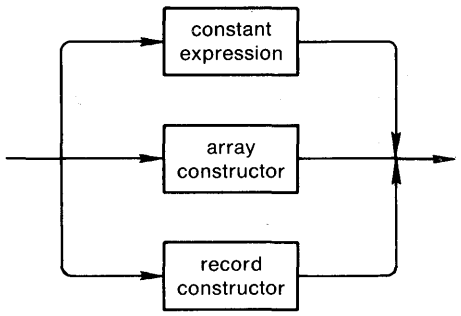
ZK-121-81

identifier



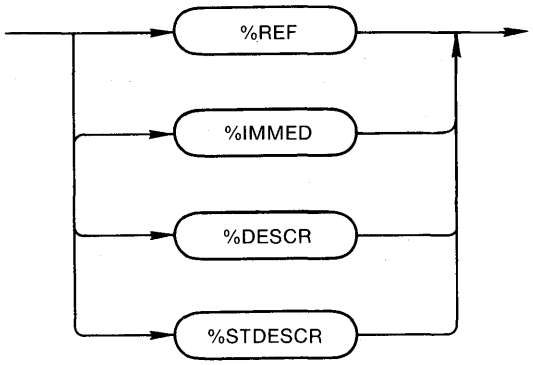
ZK-112-81

initial value



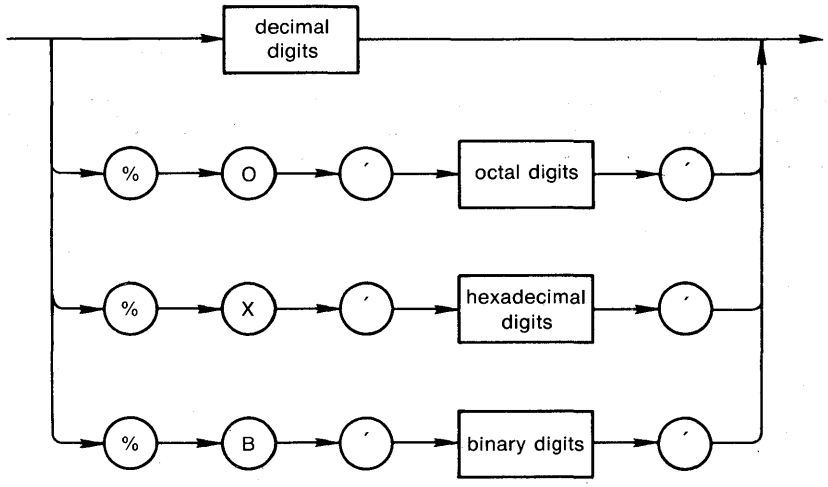
ZK-565-81

mechanism specifier



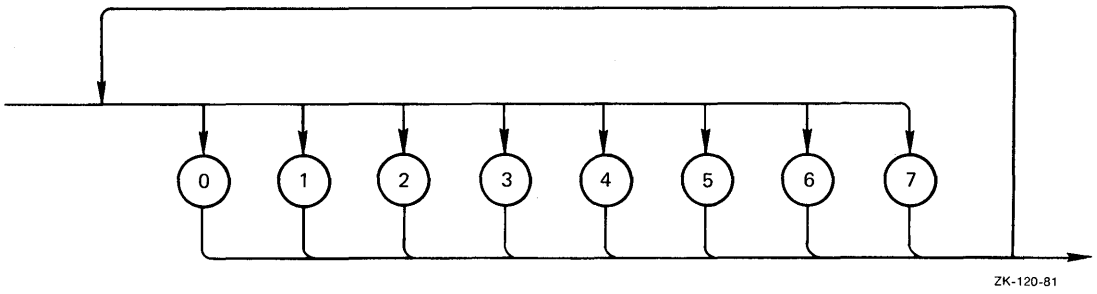
ZK-564-81

numeric constant



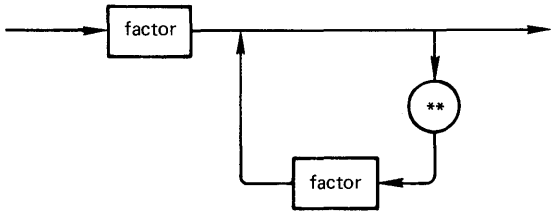
ZK-114-81

octal digits



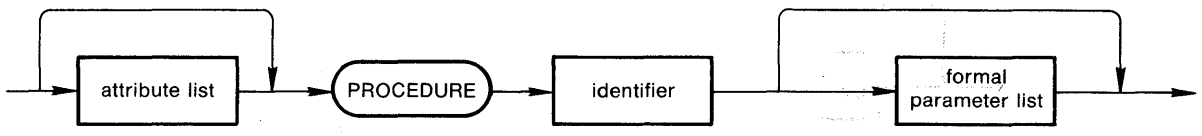
ZK-120-81

primary



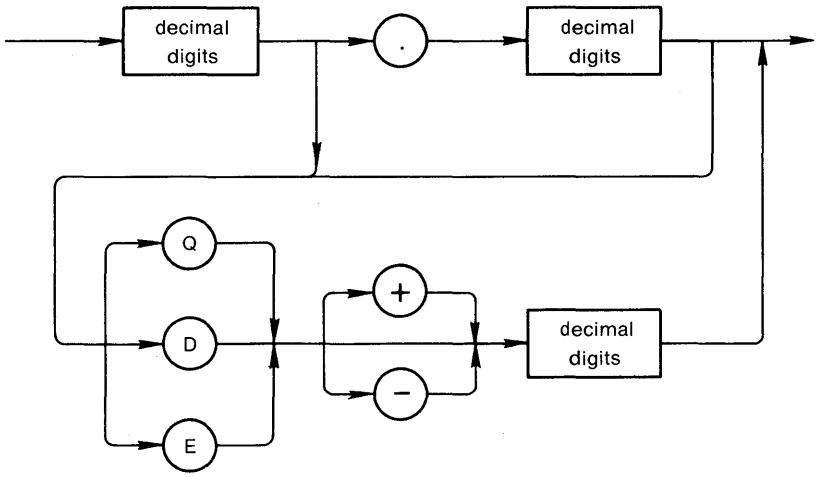
ZK-116-81

procedure heading



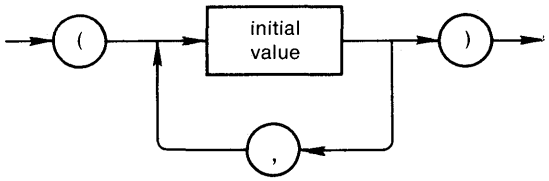
ZK-1034-82

real constant



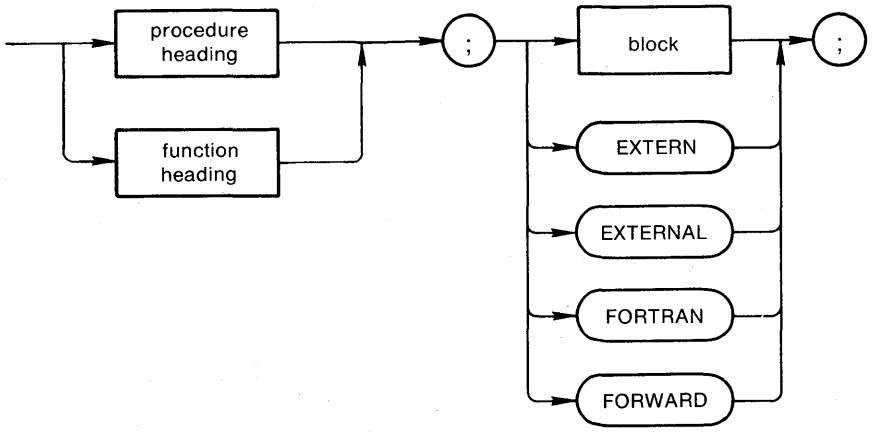
ZK-113-81

record constructor



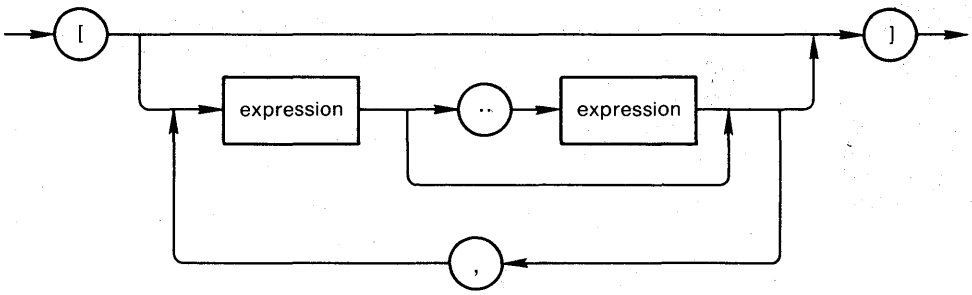
ZK-567-81

routine declaration



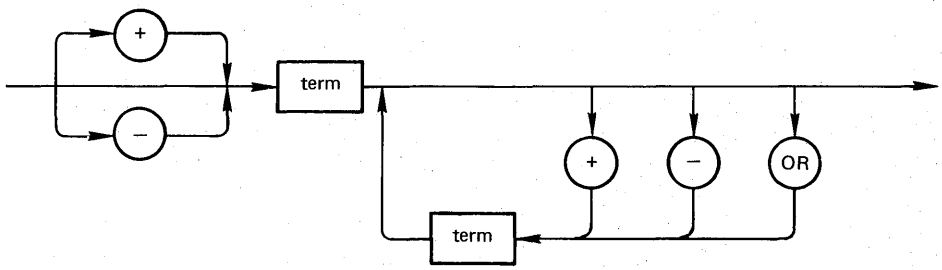
ZK-130-81

set constructor



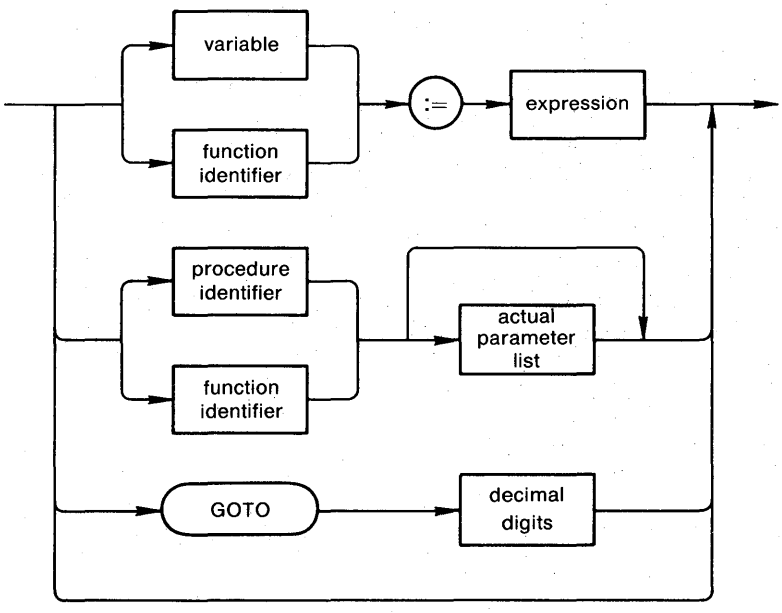
ZK-569-81

simple expression



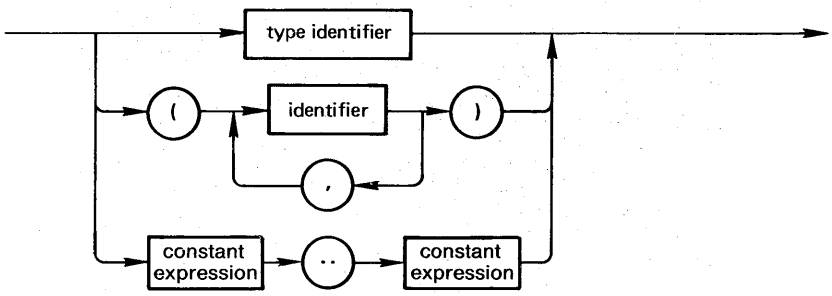
ZK-118-81

simple statement



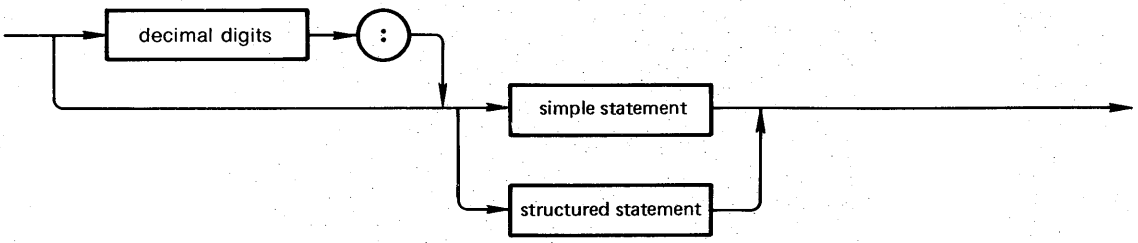
ZK-108-81

simple type



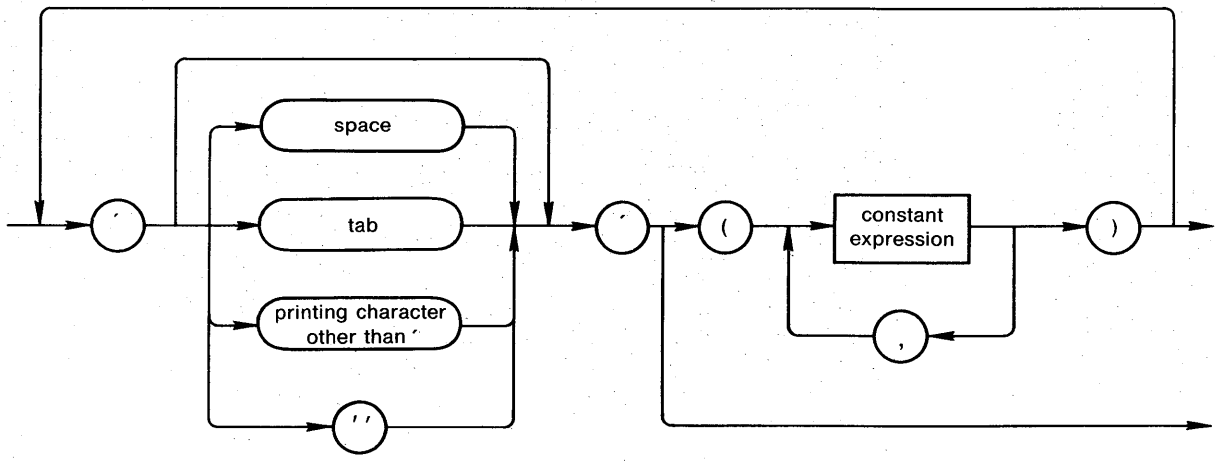
ZK-124-81

statement

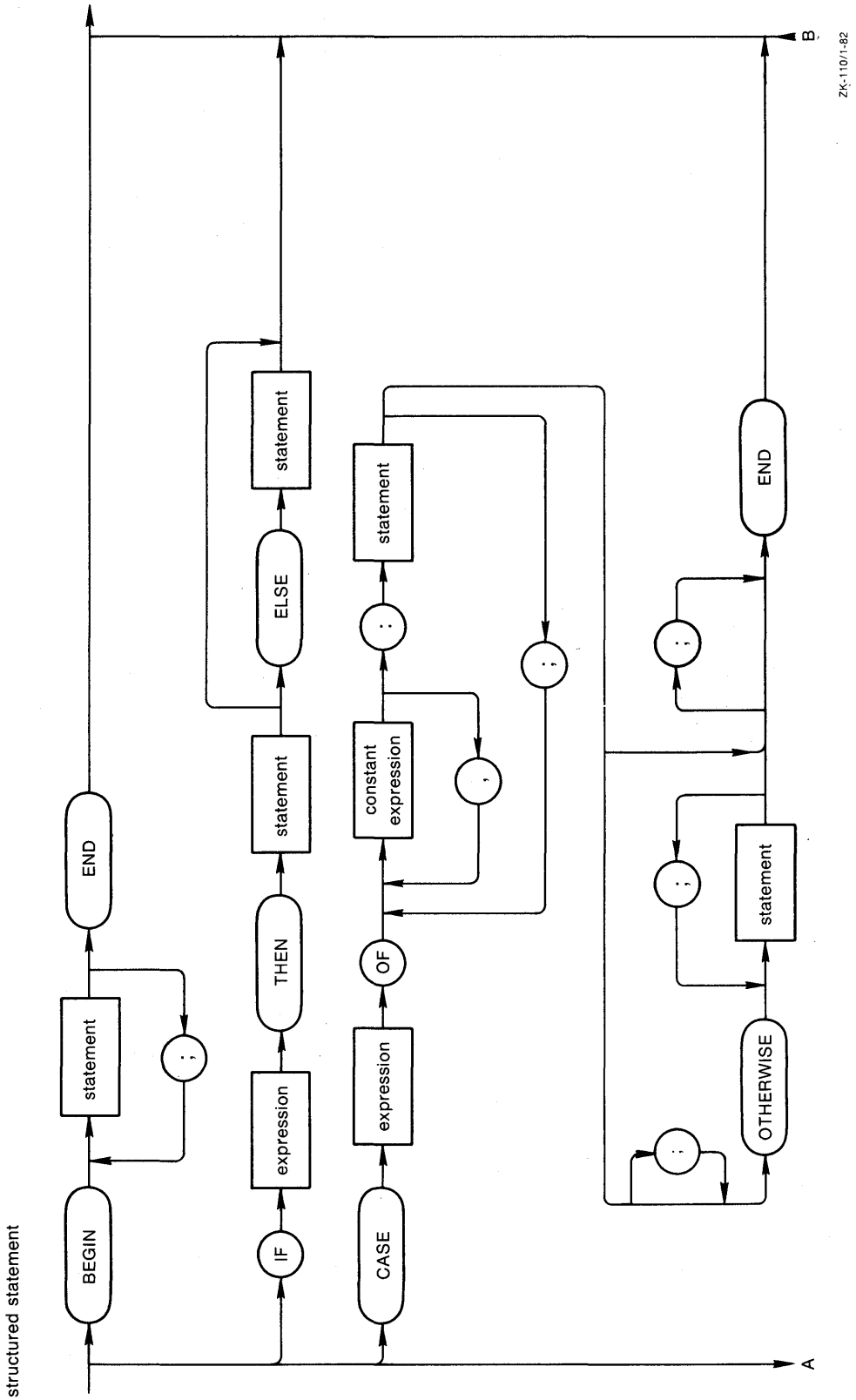


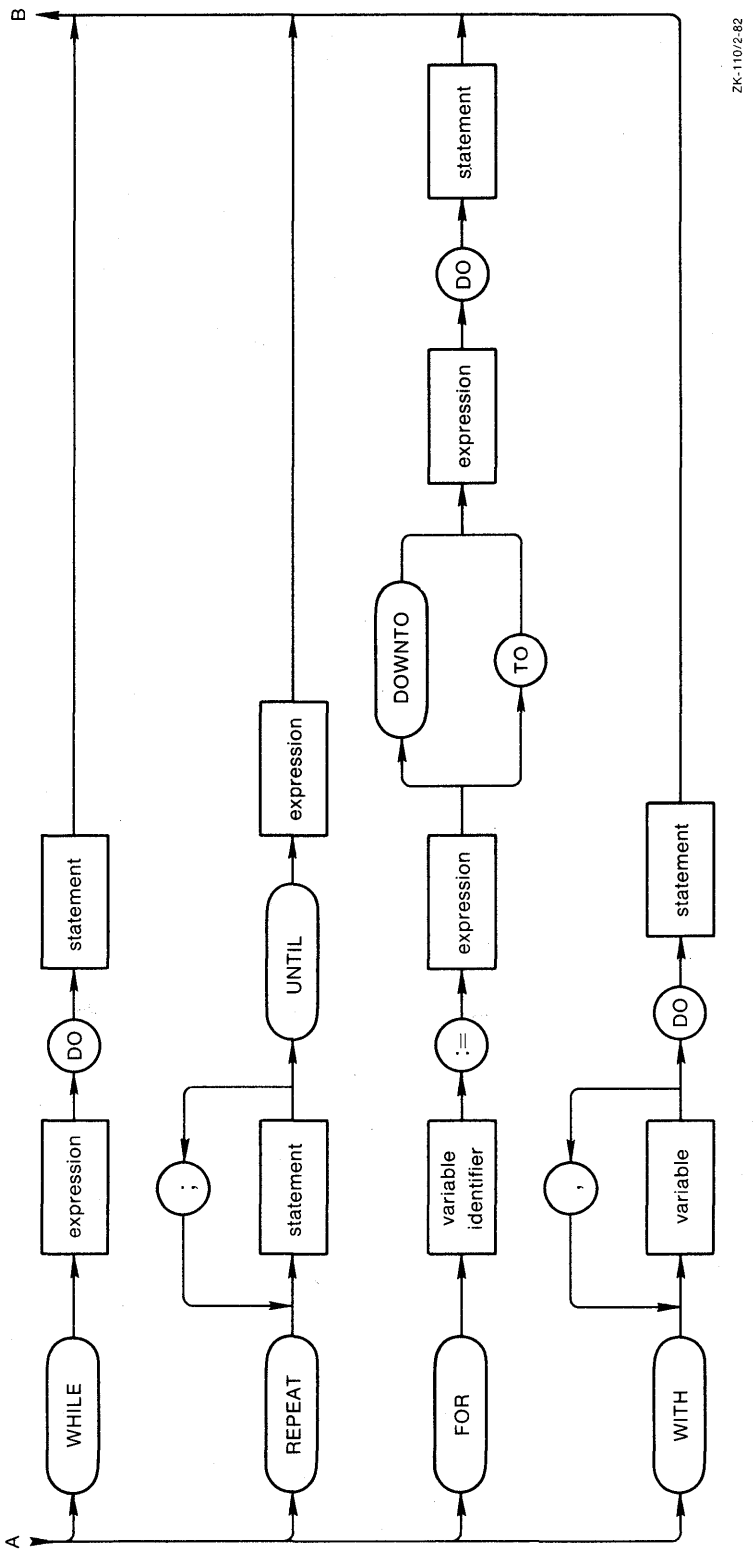
ZK-137-81

string constant

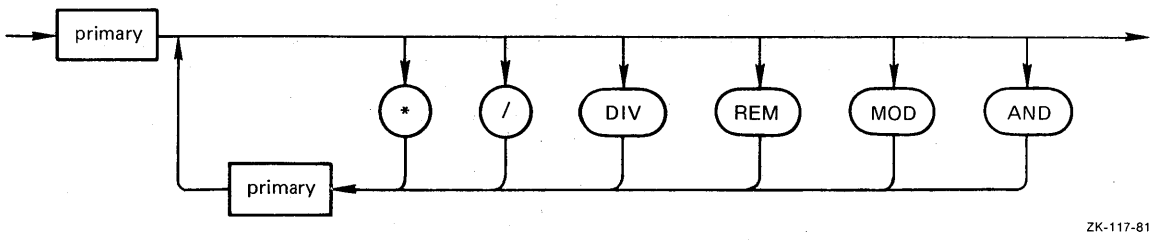


ZK-572-81


structured statement
BEGIN
statement
;
END
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant expression
,
:
statement
;
OTHERWISE
statement
;
END
A
B
ZK-110/1-82

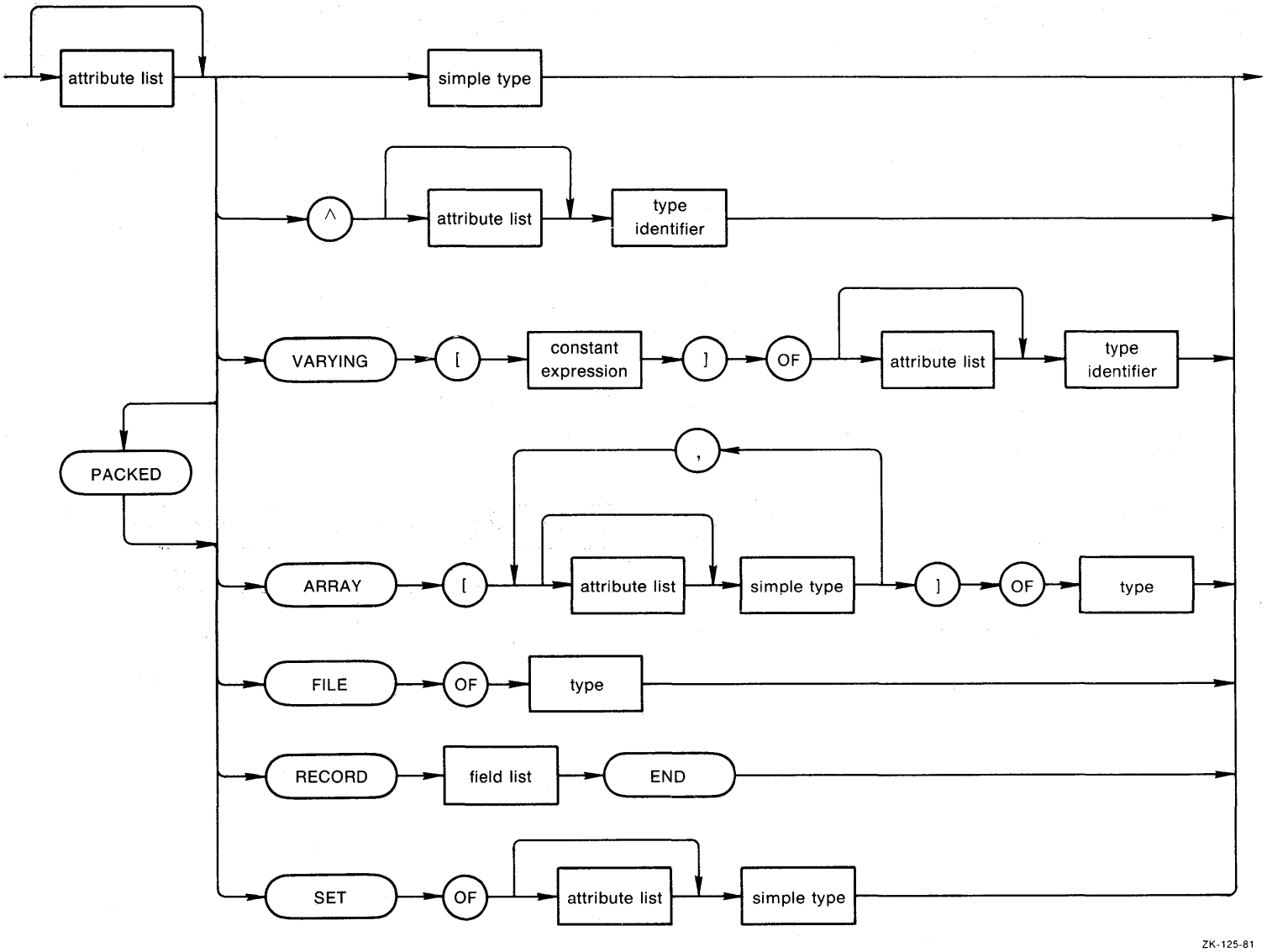

A
B
WHILE
expression
DO
statement
REPEAT
;
statement
UNTIL
expression
FOR
variable identifier
:=
expression
DOWNTO
TO
expression
DO
statement
WITH
,
variable
DO
statement
ZK-110/2-82

term

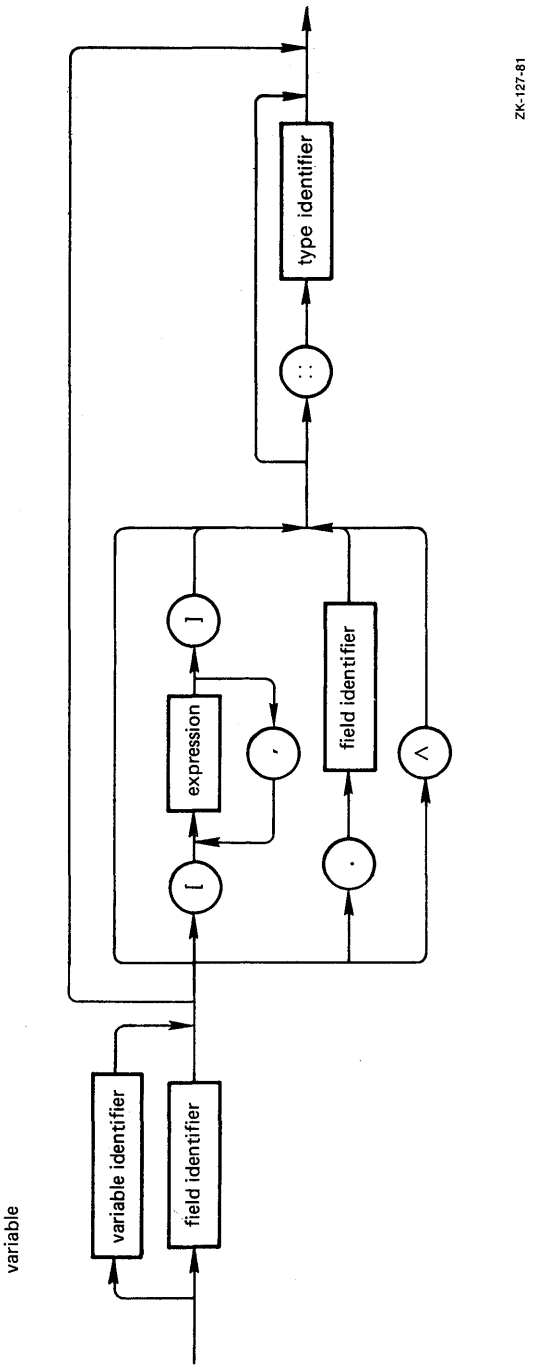


ZK-117-81

type



ZK-125-81


variable
variable identifier
field identifier
[
expression
,
]
.
field identifier
>
::
type identifier
ZK-127-81

Appendix C
Predeclared Routines

Tables C–1 and C–2 summarize VAX–11 PASCAL's predeclared procedures and functions. Routines that handle input and output are described in detail in Chapter 8; these routines can accept an optional parameter, designated here as e, that indicates the action to be taken should an error occur during the routine's execution. All other predeclared routines are described in Chapter 7.

Table C–1: Predeclared Procedures

Procedure	Parameter	Action
CLOSE(f,parameters,e)	f = file variable parameters — see the *VAX–11 PASCAL Language Reference Manual* e = error parameter	Closes file f with the specified properties.
DATE(str)	str = variable of type PACKED ARRAY [1..11] OF CHAR	Assigns current date to str.
DELETE(f,e)	f = file variable e = error parameter	Deletes the current component of file f. File f must have relative or indexed organization and be opened for direct or keyed access. The current component must be locked.
DISPOSE(p)	p = pointer value	Releases storage for p^. Any pointers to the storage become undefined.
DISPOSE(p, t1,...,tn)	p = pointer variable t1,...,tn = tag field constants	Releases storage for p^; used when p^ is a record with variants. Tag field values are optional; if specified, they must be identical to those specified when storage was allocated by NEW.
ESTABLISH(id)	id = function-identifier	Sets up a VAX–11 condition handler to process exceptions.
FIND(f,n,e)	f = file variable n = component number e = error parameter	Moves the current file position to component n of file f.

Table C–1 (Cont.): Predeclared Procedures

Procedure	Parameter	Action
FINDK(f,kn,kv,m,e)	f = file variable kn = key number kv = key value m = match type e = error parameter	Moves the current position of file f to a specified component. The match type can have a value of EQL, GTR, or GEQ to indicate that the component to be found has a value in key position kn that is equal to, greater than, or greater than or equal to key value kv. Match type m is optional and defaults to EQL. File f must be opened for keyed access.
GET(f,e)	f = file variable e = error parameter	Moves the current file position to the next component of f. Then GET(f) assigns the value of that component to f^, the file buffer variable.
HALT	None	Calls LIB$STOP, signaling PAS$_ABORT. Without an appropriate condition handler, HALT terminates execution of the program.
LOCATE(f,n,e)	f = file variable n = component number e = error parameter	Positions file f at component n so that the next PUT procedure can modify n.
LINELIMIT(f,n)	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.
NEW(p, t1,...,tn)	p = pointer variable t1,...,tn = tag field constants	Allocates storage for p^; used when p^ is a record with variants. The optional parameters t1 through tn specify the values for the tag fields of the current variant. All tag field values must be listed in the order in which they were declared. They cannot be changed during execution. NEW does not initialize the tag fields.
OPEN(f,parameters,e)	f = file variable parameters — see the *VAX-11 PASCAL Language Reference Manual* e = error parameter	Opens file f with the specified properties.
PACK(a,i,z)	a = variable of type ARRAY [m..n] OF T i = starting index of array a z = variable of type PACKED ARRAY [u..v] OF T	Moves (v–u+1) components from array a to array z by assigning components a[i] through a[i+v–u] to z[u] through z[v]. The upper bound of a must be greater than or equal to (i+v–u).
PAGE(f,e)	f = text file variable e = error parameter	Skips to the next page of file f. The next line written to f begins on the second line of a new page.
PUT(f,e)	f = file variable e = error parameter	Writes the value of f^, the file buffer variable, into the file f and moves the current file position to the next component of f.

Table C–1 (Cont.): Predeclared Procedures

Procedure	Parameter	Action
READ(f, v1,...,vn,e)	f = file variable v1,...,vn = variables e = error parameter	Assigns successive values from the input file f to the variables v1 through vn. You must specify at least one variable (v1). The default for f is INPUT.
READLN(f, v1,...,vn,e)	f = text file variable v1,...,vn = variables e = error parameter	Performs the READ procedure, then sets the current file position to the beginning of the next line. The variables v1 through vn are optional. The default for f is INPUT.
READV(s,v1,...,vn)	s = character string expression v1,...,vn = variables	Assigns successive values from the input string s to the variables v1 through vn. You must specify at least one variable (v1).
RESET(f,e)	f = file variable e = error parameter	Enables reading from file f. RESET(f) moves the current file position to the beginning of the file f and assigns the first component of f to the file buffer variable, f^. EOF(f) is set to FALSE unless the file is empty.
RESETK(f,kn,e)	f = file variable kn = key number e = error parameter	Enables reading from file f. RESETK(f,kn) moves the current file position to the component with the lowest value in key position kn. File f must be opened for keyed access.
REVERT	None	Cancels a VAX–11 condition handler set up by ESTABLISH.
REWRITE(f,e)	f = file variable e = error parameter	Enables writing to file f. REWRITE(f) truncates the file f to zero length and sets EOF(f) to TRUE.
TIME(str)	str = variable of type PACKED ARRAY [1..11] OF CHAR	Assigns the current time to str.
TRUNCATE(f,e)	f = file variable e = error parameter	Deletes current file component and all components following it. File f must have sequential organization.
UNLOCK(f,e)	f = file variable e = error parameter	Unlocks the current file component if it is locked.
UNPACK(z,a,i)	z = variable of type PACKED ARRAY[u..v] OF T a = variable of type ARRAY [m..n] OF T i = starting index in array a	Moves (v–u+1) components from array z to array a by assigning components z[u] through z[v] to a[i] through a[i+v–u]. The upper bound of a must be greater than or equal to (i+v–u).
UPDATE(f,e)	f = file variable e = error parameter	Writes the contents of the file buffer into the current component. File f must have relative or indexed organization and be opened for direct or keyed access. The current component must be locked.

Table C-1 (Cont.): Predeclared Procedures

Procedure	Parameter	Action
WRITE(f,p1,...,pn,e)	f = file variable p1,...,pn = write parameters e = error parameter	Writes the values of p1 through pn into the file f. At least one parameter (p1) must be specified. The default for f is OUTPUT.
WRITELN(f,p1,...,pn,e)	f = text file variable p1,...,pn = write parameters e = error parameter	Performs the WRITE procedure, then skips to the beginning of the next line. The write parameters are optional. The default for f is OUTPUT.
WRITEV(s,p1,...,pn)	s = character string variable p1,...,pn = write parameters	Writes the values of p1 through pn into the character strings.

Table C-2: Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
Arithmetic	ABS(x)	Any arithmetic type	Same as x	Computes the absolute value of x.
	ARCTAN(x)	INTEGER REAL	REAL	Computes the arc tangent of x. The result is expressed in radians.
	COS(x)	INTEGER REAL	REAL	Computes the cosine of x. The parameter is expressed in radians.
	EXP(x)	INTEGER REAL	REAL	Computes e**x, the exponential function.
	LN(x)	INTEGER REAL	REAL	Computes the natural logarithm of x. The value of x must be greater than 0.
	SIN(x)	INTEGER REAL	REAL	Computes the sine of x. The parameter is expressed in radians.
	SQR(x)	Any arithmetic type	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, an error occurs.
Ordinal	PRED(x)	Any ordinal type	Same as x	Returns the predecessor value in the type of x (if a predecessor exists).
	SUCC(x)	Any ordinal type	Same as x	Returns the successor value in the type of x (if a successor exists).
Boolean	EOF(f)	File variable	BOOLEAN	Indicates whether the file position is at the end of the file f. EOF(f) becomes TRUE only when the file position is after the last component in the file. The default for f is INPUT.

Table C–2 (Cont.): Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
	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 integer x is odd; returns FALSE if x is even.
Transfer	CHR(x)	INTEGER	CHAR	Returns the character (if one exists) whose ordinal value is x.
	INT(x)	Any ordinal type	INTEGER	Converts the value of x to an integer.
	ORD(x)	Any ordinal type	INTEGER	Returns the ordinal value corresponding to the value of x.
	ROUND(x)	REAL	INTEGER	Rounds the REAL value x to the nearest integer.
	TRUNC(x)	REAL	INTEGER	Truncates the REAL value x to an integer.
Dynamic Allocation	ADDRESS(x)	Any variable except a component of a packed structured type	Pointer	Returns the pointer that references the parameter.
Character String	BIN(x,l,d)	x = any type l = integer d = integer	Varying	Converts a parameter x to its binary representation. Returns binary value in a string of length l with d significant digits. Parameters l and d are optional.
	HEX(x,l,d)	x = any type l = integer d = integer	Varying	Converts a parameter x to its hexadecimal representation. Returns hexadecimal value in a string of length l with d significant digits. Parameters l and d are optional.
	INDEX(s1,s2)	Any string type	Integer	Locates first occurence of s2 within s1. Returns integer indicating leftmost position of s2. Returns 0 if s2 is not found.
	LENGTH(s)	Any string type	Integer	Returns integer value indicating current length of s.
	OCT(x,l,d)	x = any type l = integer d = integer	Varying	Converts a parameter x to its octal representation. Returns octal value in a string of length l with d significant digits. Parameters l and d are optional.
	PAD(s,fill,l)	s = string fill = character l = integer	Varying	Pads a string s with fill character until it is of length l.
	SUBSTR(s,b,l)	s = string b = integer l = integer	Varying	Constructs a new string beginning at position b of a given string s and extending to length l.

Table C–2 (Cont.): Predeclared Functions

Category	Function	Parameter Type	Result Type	Purpose
Unsigned	UAND(u1,u2)	Unsigned	Unsigned	Performs a binary logical AND on the corresponding bits of parameters u1 and u2.
	UNOT(u)	Unsigned	Unsigned	Performs a binary logical NOT on the corresponding bits of parameter u.
	UOR(u1,u2)	Unsigned	Unsigned	Performs a binary logical OR on the corresponding bits of parameters u1 and u2.
	UXOR(u1,u2)	Unsigned	Unsigned	Performs a binary logical exclusive OR on the corresponding bits of parameters u1 and u2.
Allocation Size	SIZE(x,c1,...,cn)	x = any type c1,...,cn = case constants	Integer	Returns an integer value indicating number of bytes allocated for a variable or record field of type x. If the variable is part of a variant record, case constants c1 through cn may be specified.
	NEXT(x)	Any type	Integer	Returns integer value indicating number of bytes allocated for a component of type x in an unpacked array.
	PSIZE(x)	Any type	Integer	Returns integer value indicating number of bits allocated for a field of type x in a packed record
	PNEXT(x)	Any type	Integer	Returns integer value indicating number of bits allocated for a component of type x in a packed array.
Low_Level Interlocked	ADD_INTER-LOCKED(e,v)	e = assignment compatible with v v = Integer, Unsigned, or Subrange	Integer	Adds e to v. Returns -1 if result is negative, 0 if result is zero, +1 if result is positive.
	SET_INTER-LOCKED(b)	Boolean	Boolean	Assigns TRUE to parameter b and returns its original value.
	CLEAR_INTER-LOCKED(b)	Boolean	Boolean	Assigns FALSE to parameter b and returns its original value.
Miscellaneous	CARD(s)	Set	Integer	Returns the number of elements currently belonging to the set s.
	CLOCK	None	Integer	Returns an integer value equal to the central processor time used by the current process. The time is expressed in milliseconds.
	EXPO(r)	Real, Double, Quadruple	Integer	Returns the integer-valued exponent of the floating-point representation of r.
	STATUS(f)	File	Integer	Returns 0 if the previous operation on the file succeeded, -1 if the previous operation encountered an end-of-file, and a positive integer representing an error code if the previous operation resulted in an error.

Appendix D
Summary of VAX-11 PASCAL Extensions

Table D-1 summarizes the language features provided in VAX-11 PASCAL that are not part of the PASCAL language definition.

Table D-1: Language Extensions

Category	Extension
Lexical and syntactical extensions	Reserved words: MODULE, OTHERWISE, REM, VARYING, %DESCR, %STDESCR, %IMMED, %REF, %INCLUDE
	Exponentiation operator (**)
	REM operator
	Type cast operator (::) for variables and expressions
	Binary, hexadecimal, and octal notation for integers
	Double- and quadruple-precision real numbers
	Dollar sign ($) and underscore (_) characters in identifiers
	Identifiers that can begin with any character other than a digit but whose first 31 characters must be unique
	Extended syntax for inclusion of nonprinting characters in character strings
	Compile-time constant expressions allowed anywhere a constant is allowed
	Constructors of structured types used anywhere in place of a constant of the structured type
	Attributes used with data items, routines, and compilation units
	Relaxed rules for assignment compatibility
	Structural compatibility enforces between actual and formal parameters

Table D-1 (Cont.): Language Extensions

Category	Extension
Predefined types	UNSIGNED, SINGLE, DOUBLE (D_floating and G_floating), QUADRUPLE
	VARYING OF CHAR structured type and concatenation operator for VARYING strings
Predeclared procedures	CLOSE, DATA, DELETE, ESTABLISH, FIND, FINDK, HALT, LINELIMIT, LOCATE, OPEN, READY, RESETK, REVERT, TIME, TRUNCATE, UNLOCK, UPDATE, WRITEV
Predeclared functions	Boolean functions: UFB and UNDEFINED
	Transfer functions: DBLE, INT, QUAD, TRUNC, UINT, UROUND, UTRUNC
	Dynamic allocation function: ADDRESS
	Character-string functions: BIN, HEX, INDEX, LENGTH, OCT, PAD, SUBSTR
	Unsigned functions: UAND, UNOT, UOR, UXOR
	Allocation size functions: SIZE, NEXT, BITSIZE, BITNEXT
	Low-level interlocked functions: ADD_INTERLOCKED, SET_INTERLOCKED, CLEAR_INTERLOCKED
	Miscellaneous functions: CARD, CLOCK, EXPO
READ, READLN, WRITE, WRITELN extensions	Parameters of character-string and enumerated types for READ and READLN
	Parameters of enumerated types for WRITE AND WRITELN
	Prompting at the terminal with a WRITE/READ or WRITE/READLN sequence
	Optional carriage-control specification for text files with WRITE and WRITELN
Extended I/O capabilities	Direct access and relative file organization
	Keyed access and indexed file organization
Declarations	Declaration and definition sections that can appear more than once and in any order
	Initialization of static variables in VAR declaration sections at program or module level
	VALUE initialization section
Statements	OTHERWISE clause in CASE statement

Table D–1 (Cont.): Language Extensions

Category	Extension
Procedures and functions	Functions that return values of structured types (other than file types)
	Function called as procedures
	External procedure and function declarations
	Default values for formal parameters
	Nonpositional parameter passing
	Extended mechanism specifiers for passing parameters to external procedures and functions: %IMMED, %REF, %DESCR, %STDESCR
Compilation	MODULE capability for combining declarations and definitions to be compiled independently from the main program
	ENVIRONMENT and INHERIT attributes to control independent compilation

Appendix E
Differences Between Version 1 and Version 2

This appendix describes the differences between previous versions of VAX-11 PASCAL and Version 2. These differences fall into three categories:

- Features that have been decommitted. The previous versions of these features are still supported in Version 2 to allow you to run existing programs; however, it is recommended that you modify your programs to reflect the Version 2 changes.
- Features that are controlled by the /OLD_VERSION compile-time qualifier.
- Minor changes that are not likely to affect the vast majority of existing VAX-11 PASCAL programs.

If you modify a program that executed successfully under previous versions of VAX-11 PASCAL, you should not make changes that conflict with the Version 2 language definition. If conflicts exist and you compile the program with Version 2, one of two problems may result:

- You may get warning messages at compile time.
- The program may compile successfully but may not run.

If you must use language features that conflict with Version 2, you can use the /OLD_VERSION qualifier at compile time to produce the desired results. The /OLD_VERSION qualifier and the conflicts that it resolves are described in Section E.2.

E.1 Decommitted Features

The following decommitted features are described in this section:

- VALUE initialization section
- Syntax of dynamic array parameters
- Predeclared functions LOWER and UPPER
- Printing of hexadecimal and octal values with the WRITE procedure
- Syntax of the OPEN procedure
- Specification of compiler qualifiers in the source code

E.1.1 VALUE Section

The VALUE section initializes variables that are declared in the main program declaration section. You can initialize ordinal, real, and structured variables (except file variables) with constants or constructors of the same type.

The description below presents general information on VALUE initializations. The exact format of the initialization depends on the type of the variable being initialized. For more information on types, refer to Chapter 2.

Syntax

VALUE {variable-identifier := value};...

variable-identifier

The name of the variable to be initialized. You cannot specify a list of variable identifiers.

value

A constant of the same type as the variable, or a constructor for a record, array, or set variable.

You must specify a constant of the correct type for each variable being initialized; you cannot specify an expression. Note that structured variables require constructors (see Chapter 2).

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

VAX-11 PASCAL Version 2 allows you to initialize variables in a VAR declaration section of the main program (see Section 4.4).

E.1.2 Dynamic Array Parameters

Some programming applications require general routines that can process arrays with different bounds. Version 1 of VAX-11 PASCAL allows you to declare routines with dynamic array parameters. You can call the routine with arrays of different sizes, as long as their bounds are within those specified by the formal parameter.

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

In routines 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 (see Section E.1.3).

Syntax

{array-identifier},... : [PACKED] ARRAY[{index-type-identifier},...]
OF type-identifier

Note that you must use a type identifier to specify the range of the indexes. You cannot use a subrange. The type identifier can be any of the predefined ordinal types (for example, INTEGER).

The components and indexes of the actual and formal dynamic array parameters must be of compatible types. The rules for dynamic array compatibility (see Section E.1.2) are identical to those for compatibility between other arrays, with one exception: the range of the index types of the actual array parameter must be within the range specified for the formal parameter.

The following differences exist between the Version 2 syntax and the syntax of previous versions. See Section 6.3.5 for further details of the Version 2 syntax.

- In Version 2, dynamic arrays are known as conformant arrays, and the syntax that describes them is called a conformant array schema.
- The conformant array schema for Version 2 requires that the upper and lower bounds of the conformant array parameter be declared with identifiers in the formal parameter list. You can then use these identifiers within the routine block to refer to the upper and lower bounds of the parameter.
- Version 2 allows the type identifier of a conformant array parameter to be another conformant array schema.

E.1.3 Lower and Upper Functions

Previous versions of VAX-11 PASCAL included the predeclared functions LOWER and UPPER, which you could use to determine the upper and lower bounds of dynamic array parameters (see Section E.1.2). Because the syntax of conformant array parameters has changed for Version 2 (see Section 6.3.5), these functions are no longer necessary. They are supported, however, for programs that use the old syntax.

Syntax

```
LOWER (a [, n])
UPPER (a [, n])
```

The parameter a denotes an array variable; the optional parameter n is an integer constant that denotes a dimension of a. If you omit a value for the parameter n, it defaults to 1. The LOWER function returns the lower bound of the nth dimension of a; the UPPER function returns the upper bound of the nth dimension of a.

E.1.4 Printing Hexadecimal and Octal Values

The following sections explain how to print values in hexadecimal and octal notation using the WRITE procedure. Version 2 provides the predeclared functions HEX, OCT, and BIN, which return the hexadecimal, octal, and binary equivalents of the input value (see Section 7.6). You can use these functions in conjunction with the WRITE, WRITELN, and WRITEV procedures to print values in hexadecimal, octal, and binary notation.

The following formats of the WRITE procedure are used to print hexadecimal and octal values in Version 1.

Syntax

```
WRITE ({expression:field-width HEX},...)
WRITE ({expression:field-width OCT},...)
```

expression

The value to be written. Arbitrary items (including pointers) can be written in hexadecimal or octal notation to text files.

field-width

A positive integer expression indicating the length of the print field.

For hexadecimal values, if the field width specified is less than eight characters, and the output value 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 value is less than the field width, the field is padded with blanks on the left.

For octal values, if the field width specified is less than 11 characters, and the output value 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 value is less than the field width, the field is padded with blanks on the left.

Examples

1.
```
WRITE (Payroll:10 HEX);
```

   The value of the variable Payroll is printed in a field of 10 hexadecimal characters.

2.
```
WRITE (Social_Security:14 OCT);
```

   The value of the variable Social_Security is printed in a field of 14 octal characters.

E.1.5 The OPEN Procedure

The OPEN procedure opens a file and allows you to specify file parameters. Version 2 includes new parameters and additional parameter values and has changed some defaults. Table E–1 lists the file parameters available under Version 1, their possible values, and their defaults.

Table E–1: Summary of Version 1 OPEN Parameters

Parameter	Parameter Values	Default
History	OLD or NEW	NEW (OLD, if the file is opened using RESET)
Record-length	Any positive integer	133 bytes
Access-method	DIRECT or SEQUENTIAL	SEQUENTIAL
Record-type	FIXED or VARIABLE	VARIABLE for new files; for old files, 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 parameter

The following differences exist between the Version 2 OPEN syntax and the syntax of Version 1. See Section 8.3.1 for a complete description of the Version 2 OPEN procedure.

- In Version 1, the file name is specified as a string constant (VAX/VMS file specification) or a logical name. In Version 2, a string expression containing a file specification can be used as the file name.
- In Version 2, the parameter values READONLY and UNKNOWN have been added to the history parameter.
- In Version 2, the parameter value KEYED has been added to the access-method parameter.
- In Version 2, the default record type is VARIABLE for new text files and files of type FILE OF VARYING OF CHAR; for all other new files, the default is FIXED. The default for old files remains the same.
- In Version 2, the default carriage control is LIST for text files and files of type FILE OF VARYING OF CHAR. The default for all other file types and for old files remains the same.
- Version 2 includes five new parameters for the OPEN procedure: organization, disposition, sharing, user-action, and error-recovery. These parameters, their possible values, and their defaults are described in Section 8.3.1.

Note that although direct access to text files is prohibited in both Version 1 and Version 2, the point at which the error occurs differs in the two versions. In Version 1, an OPEN procedure is allowed to specify direct access for a text file; the error occurs when a FIND procedure attempts to access the file. In Version 2, an OPEN procedure that specifies direct access to a text file causes an error to be generated.

E.1.6 Specifying Qualifiers in the Source Code

In previous versions of VAX-11 PASCAL, you could specify compiler qualifiers within comments in the source code. VAX-11 PASCAL Version 2 does not support this feature. It is recommended that you specify these qualifiers with the PASCAL command when you compile the program. Alternatively, you can use attributes in your program to perform some of the same operations that are performed by compiler qualifiers. For more information, refer to Chapter 10 and to the *VAX-11 PASCAL User's Guide*.

In Version 1, the CHECK qualifier (abbreviated C) generates code to perform run-time checks. The CROSS_REFERENCE qualifier (X) produces a cross-reference listing of identifiers. The DEBUG qualifier (D) generates records for the VAX-11 Symbolic Debugger. The LIST qualifier (L) produces a source listing file. The MACHINE_CODE qualifier (M) includes machine code in the source listing file. The STANDARD qualifier (S) prints informational messages indicating the use of VAX-11 PASCAL extensions. The WARNINGS qualifier (W) prints diagnostics for warning-level errors.

Syntax

(*${qualifier},...[; comment] *)

qualifier

A qualifier name or a 1-character abbreviation.

comment

The text of a comment, which is optional.

The first character after the comment delimiter must be a dollar sign ($), which cannot be preceded by a space.

To enable a qualifier, use a plus sign (+) after the qualifier's name or abbreviation. To disable a qualifier, use a minus sign (-) after the qualifier's 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 will not be in effect.

E.2 /OLD_VERSION Qualifier

The VAX-11 PASCAL language definition in early versions conflicts in some respects with that of Version 2, which is based on Level 0 of the standard proposed by the International Organization for Standardization (ISO). The /OLD_VERSION qualifier on the PASCAL command informs the compiler that it should default to the VAX-11 PASCAL Version 1 language definition when conflicts arise. By default, /OLD_VERSION is disabled so that the compilation conforms to the PASCAL standard.

Because the Version 2 compiler performs many optimizations on the source code, you should also enable the /NOOPTIMIZE qualifier during the recompilation of old programs. The /NOOPTIMIZE qualifier prevents the compiler from making optimizations that might cause an old program to behave unexpectedly when it is recompiled.

The following sections describe the conflicts between Version 2 and the previous versions of VAX-11 PASCAL and explain how they are resolved by the /OLD_VERSION qualifier.

E.2.1 Comment Delimiters

Version 2, unlike previous versions, considers the opening comment delimiters (* and { equivalent; likewise, the closing delimiters *) and } are considered equivalent. Therefore, a comment begun with (* can be terminated with }, and a comment begun with { can be terminated by *).

Recompilation of the program with the /OLD_VERSION qualifier will cause the Version 1 restriction to be enforced so that you cannot combine comment delimiters in this way.

E.2.2 %INCLUDE Files

In previous versions of VAX-11 PASCAL, the default file type of a %INCLUDE file is DAT. However, in Version 2, the default file type is PAS.

You must use the /OLD_VERSION qualifier to recompile a program that includes one or more files that have a file type of DAT.

E.2.3 Multidimensional Packed Arrays

Previous versions of the VAX-11 PASCAL compiler interpret the shorthand form of the array type definition

```
PACKED ARRAY[x,y,z]
```

to be equivalent to the longer definition

```
ARRAY[x] OF ARRAY[y] OF PACKED ARRAY[z]
```

That is, only the last dimension of the array is packed. In Version 2, however, the shorthand definition above is equivalent to the longer definition:

```
PACKED ARRAY[x] OF PACKED ARRAY[y] OF PACKED ARRAY[z]
```

In the Version 2 interpretation, all dimensions of the array are packed.

You must use the /OLD_VERSION qualifier to recompile a program that includes a multidimensional packed array of which you want only the last dimension to be packed.

E.2.4 Storage of Components

In previous versions of VAX-11 PASCAL, a component of the subrange type 0..0 in a packed record or array is allocated one bit of storage. In Version 2, however, a component of this type is not allocated any storage.

You must use the /OLD_VERSION qualifier to recompile a program in which one bit of storage is required to be allocated for such a component.

E.2.5 Storage of Sets

In previous versions of VAX-11 PASCAL, an unpacked set type was always allocated 256 bits. In Version 2, the allocation size of an unpacked set depends on the set's base type and on whether the unpacked set is allocated in a packed or an unpacked context. See the *VAX-11 PASCAL User's Guide* for details.

You must use the /OLD_VERSION qualifier to recompile a program in which an unpacked set requires an allocation size of 256 bits.

E.2.6 TEXT Files and FILE OF CHAR

Previous versions of VAX-11 PASCAL consider the predefined file types TEXT and FILE OF CHAR to be equivalent. In Version 2, however, files of type TEXT are composed of complete lines of characters terminated by an end-of-line marker, while files of type FILE OF CHAR are composed of individual characters. Section 8.1.1 and the *VAX-11 PASCAL User's Guide* provide detailed information about the differences between these two file types.

You must use the /OLD_VERSION qualifier to recompile a program that requires a TEXT file and a FILE OF CHAR to be treated identically.

E.2.7 MOD Operator

The MOD operator, as defined by previous versions of VAX-11 PASCAL, returns the remainder that results from the DIV operation. In Version 2, however, the MOD operator is equivalent to the mathematical modulus operation. Therefore, Version 2 allows you to perform the operation I MOD J only when J is a positive number; the MOD function always returns a value from 0 to J-1. To compute the remainder from the DIV operation, Version 2 provides the REM operator. (See Section 3.2.1 for more information about the MOD and REM operators.)

You must use the /OLD_VERSION qualifier to recompile a program in which you use the MOD operator to compute the remainder.

E.2.8 String Variable Parameters to the READ Procedure

In previous versions of VAX-11 PASCAL, if a READ procedure encounters an end-of-line marker as the first character to be read into a string variable, it ignores the marker and advances the file position to the beginning of the next line of input. In Version 2, a READ procedure never skips an end-of-line marker that is the first character to be read into a string variable. If a READ procedure encounters an initial end-of-line, the file remains positioned at the end of the line; you must call a READLN procedure to advance the file position to the next line. See Section 8.4.2 for further discussion of the READ procedure with string variable parameters.

Recompilation with the /OLD_VERSION qualifier causes a READ procedure to skip one end-of-line marker that it encounters as the first character to be read into a string variable.

E.2.9 Field Widths

In previous versions of VAX-11 PASCAL, a value of type REAL or SINGLE is written with a default field width of 16 characters; a value of type DOUBLE, with 24. In Version 2, however, the default field width for a value of type REAL or SINGLE is 12 characters; for a value of type DOUBLE, 20.

In addition, previous versions of VAX-11 PASCAL always expand the field width of a real number written in decimal format (when necessary) so that the real number is preceded by a leading blank. No leading blank is inserted in Version 2.

You must use the /OLD_VERSION qualifier to recompile a program in which you want to use the default field width specifications of Version 1.

E.2.10 Global Identifiers

In previous versions of VAX-11 PASCAL, the names of program-level procedures and functions are considered global identifiers. However, in Version 2, such names are not considered global unless they have the GLOBAL attribute.

You must use the /OLD_VERSION qualifier to recompile a program in which the names of program-level routines are to be made global by default.

E.2.11 Allocation in Program Sections

Unlike previous versions, Version 2 of VAX-11 PASCAL does not allocate storage for static, program-level variables in an overlaid program section. (See the *VAX-11 PASCAL User's Guide* for information about program section allocation in Version 2.)

To cause the Version 2 compiler to treat static, program-level variables and routine identifiers in the same manner as previous versions, you must use the /OLD_VERSION qualifier. You can also apply the OVERLAID attribute (see Section 10.14) to a compilation unit to cause the storage for its static, program-level variables to be allocated in an overlaid program section. Enabling /OLD_VERSION has the same effect as applying the OVERLAID attribute to all compilation units in a program.

E.3 Minor Language Changes

Some minor language changes that have been made in Version 2 cannot be controlled by the /OLD_VERSION qualifier. Such changes, however, are not likely to have adverse affects on most existing VAX-11 PASCAL programs. These changes are as follows:

- To flag language extensions when the /STANDARD qualifier is enabled, Version 2 uses the PASCAL standard proposed by the International Organization for Standardization as the language definition. The Version 1 language is defined by the *PASCAL User Manual and Report* by Jensen and Wirth.
- In Version 2, the /STANDARD qualifier is disabled by default. The Version 2 compiler does not automatically flag extensions to the PASCAL language definition contained in the ISO standard. /STANDARD was enabled in Version 1.
- Version 2 ignores all comments whose first character (inside the opening delimiter) is a dollar sign ($). Note that this behavior prohibits the specification of compile-time qualifiers in the source code, which was legal in Version 1 (see Section E.1.6).
- In Version 2, the /CHECK qualifier is enabled by default to check the bounds of array and character-string assignments. You can change the default if you wish, and you can also specify the checking of other aspects of your program. /CHECK was disabled by default in Version 1 and did not allow you to specify checking options.
- In Version 2, a change in the value of the control variable inside the body of a FOR statement does not affect the number of times the loop body is executed. (This behavior is the reverse of Version 1.)
- Version 2 considers EOLN to be FALSE when EOF is TRUE. (In Version 1, EOLN was TRUE at end-of-file.) This change is necessary to make Version 2 conform to the ISO standard, which forbids a program from testing for EOLN at end-of-file.
- A negative field-width value in a WRITE or WRITELN procedure call generates an error in Version 2. In Version 1, a negative field-width value defaulted to 0.
- When writing double-precision values, Version 2 output procedures indicate the exponent by the letter E. (Version 1 used the letter D on output values.) Note, however, that input procedures in both Version 1 and Version 2 accept either D or E as the exponent letter of double-precision values.
- In Version 2, the default length of a record in a text file is 133 bytes. Because of an error in Version 1, the default length was actually 254, contrary to the description in the documentation.
- LIB$ESTABLISH, the Run-Time Library procedure that sets up condition handlers, cannot be used in Version 2. Instead, you use the new predeclared procedures ESTABLISH and REVERT.

- Run-time condition values have new values in Version 2. These values are contained in the file SYS$LIBRARY:PASDEF.PAS. To make them available in your program, include the file in a CONST section.
- In Version 2, when a nonlocal GOTO statement transfers control from a routine to a labeled statement in an enclosing block, any condition handlers established by intervening routines are called first with the condition SS$_UNWIND. In Version 1, a nonlocal GOTO statement transferred control directly to the labeled statement; no condition handlers were executed for intervening routines.
- In Version 2, you cannot use the predeclared functions SNGL and ORD as function parameters using the Version 1 syntax for function parameter declarations. You must rewrite the formal parameter declarations to use the Version 2 syntax (see Section 6.3.3).

Appendix F
Error Detection

This appendix describes how the VAX-11 PASCAL compiler and run-time system detect violations of the PASCAL language standard. Errors detected at run time cause a program to terminate and return appropriate error messages. Errors described here as "not detected" cause a program to produce unexpected results.

The type of an index value is not assignment compatible with the index type of an array.

Explanation: Detected at run time if bounds checking was enabled during compilation.

The current variant changes while a reference to it exists.

Explanation: Not detected. An example of a reference to a variant is the passing of the variant to a formal VAR parameter.

The value of a variable to which a pointer refers (p^) is NIL.

Explanation: Usually detected at run time. Always detected if pointers checking was enabled during compilation.

The value of a variable to which a pointer refers (p^) is undefined.

Explanation: Not detected.

The DISPOSE procedure is called to dispose of a heap-allocated variable while a reference to the variable exists.

Explanation: Not detected. Examples of such references are: passing the variable or a component of it to a formal VAR parameter, or using the variable in a WITH statement (if the variable is a record).

The value of file f changes while a reference to f^ exists.

Explanation: Not detected. An example of a reference to f^ is the passing of f^ by reference to a routine; until the routine has ceased execution, you may not perform any operation on file f.

The ordinal type of an actual parameter is not assignment compatible with the type of the corresponding formal parameter.

Explanation: Detected at run time if subrange checking was enabled during compilation of the called routine.

The set type of an actual parameter is not assignment compatible with the type of the corresponding formal parameter.

Explanation: Detected at run time if subrange checking was enabled during compilation of the called routine.

A file is not in Generation mode when a PUT, WRITE, WRITELN, or PAGE procedure is attempted.

Explanation: Detected at run time.

A file is in Undefined mode when a PUT, WRITE, WRITELN, or PAGE procedure is attempted.

Explanation: Not detected.

The result of an EOF function is not TRUE when a PUT, WRITE, WRITELN, or PAGE procedure is attempted.

Explanation: Detected at run time. The operation is illegal only when the file is accessed sequentially.

The value of the file buffer variable is undefined when a PUT procedure is attempted.

Explanation: Not detected.

A file is in Undefined mode when a RESET procedure is attempted.

Explanation: Not detected.

A file is not in Inspection mode when a GET, READ, or READLN procedure is attempted.

Explanation: Detected at run time.

A file is in Undefined mode when a GET, READ, or READLN procedure is attempted.

Explanation: Not detected.

The result of an EOF function is TRUE when a GET, READ, or READLN procedure is attempted.

Explanation: Detected at run time.

The type of the file buffer variable is not assignment compatible with the type of the variable that is a parameter to a READ or READLN procedure.

Explanation: Detected at run time.

The type of the expression being written by a WRITE or WRITELN procedure is not assignment compatible with the type of the file buffer variable.

Explanation: Detected at run time.

The current variant does not exist in the list of variants specified with the NEW procedure.

Explanation: Not detected.

The DISPOSE(p) procedure is called to deallocate a pointer variable that was created using the variant form of the NEW procedure.

Explanation: Not detected.

The variant form of the DISPOSE procedure does not specify the disposal of the same number of variants that were created by the variant form of the NEW procedure.

Explanation: Not detected.

The variant form of the DISPOSE procedure does not specify the disposal of the same variants that were created by the variant form of the NEW procedure.

Explanation: Not detected.

The value of the parameter to the DISPOSE procedure is NIL.

Explanation: Detected at run time.

The value of the parameter to the DISPOSE procedure is undefined.

Explanation: Not detected.

A variant record created by the NEW procedure is accessed as a whole, rather than one component at a time.

Explanation: Not detected.

In the PACK(a,i,z) procedure, the type of the index value i is not assignment compatible with the index type of a.

Explanation: Detected at run time if subrange checking was enabled during compilation.

The PACK procedure is attempted when the value of at least one component of a is undefined.

Explanation: Not detected.

The index value i in the PACK procedure is greater than the upper bound of the index type of a.

Explanation: Detected at run time.

In the UNPACK(z,i,a) procedure, the type of the index value i is not assignment compatible with the index type of a.

Explanation: Detected at run time if subrange checking was enabled during compilation.

The index value i in the UNPACK procedure is greater than the upper bound of the index type of a.

Explanation: Detected at run time if subrange checking was enabled during compilation.

The UNPACK procedure is attempted when the value of at least one component of z is undefined.

Explanation: Not detected.

The resulting value of SQR (X) does not exist.

Explanation: Detected at run time for integers if overflow checking was enabled during compilation; always detected at run time for real numbers.

In the expression LN (X), the value of X is negative.

Explanation: Detected at run time.

In the expression SQRT (X), the value of X is negative.

Explanation: Detected at run time.

The resulting value of TRUNC (X) does not exist after the following calculations have been done: if the value of X is positive or zero, then 0 <= X-TRUNC (X) <1; otherwise, -1 < X-TRUNC (X) <=0.

Explanation: Detected at run time if overflow checking was enabled during compilation.

The resulting value of ROUND (X) does not exist after the following calculations have been done: if the value of X is positive or zero, ROUND (X) is equivalent to TRUNC (X+ 0.5); otherwise, ROUND (X) is equivalent to TRUNC (X-0.5).

Explanation: Detected at run time if overflow checking was enabled during compilation.

The resulting value of CHR (X) does not exist.

Explanation: Detected at run time if subrange checking was enabled during compilation.

The resulting value of SUCC (X) does not exist.

Explanation: Detected at run time if subrange checking was enabled during compilation.

The resulting value of PRED (X) does not exist.

Explanation: Detected at run time if subrange checking was enabled during compilation.

The function EOF (f) is called when the file f is undefined.

Explanation: Not detected.

The function EOLN (f) is called when the file f is undefined.

Explanation: Not detected.

The function EOLN (f) is called when the result of EOF (f) is TRUE.

Explanation: Not detected.

A variable is not initialized before it is first used.

Explanation: Not detected.

In the expression X/Y, the value of Y is zero.

Explanation: Detected at run time.

In the expression I DIV J, the value of J is zero.

Explanation: Detected at run time.

In the expression I MOD J, the value of J is zero or negative.

Explanation: Detected at run time if subrange checking was enabled during compilation.

An operation or function involving integers does not conform to the mathematical rules for integer arithmetic.

Explanation: Detected at run time if overflow checking was enabled during compilation.

A function result is undefined when the function returns control to the calling block.

Explanation: Not detected.

The ordinal type of an expression is not assignment compatible with the type of the variable or function identifier to which it is assigned.

Explanation: Detected at run time if subrange checking was enabled during compilation.

The set type of an expression is not assignment compatible with the type of the variable or function identifier to which it is assigned.

Explanation: Detected at run time if subrange checking was enabled during compilation.

None of the case labels is equal in value to the case selector in a CASE statement.

Explanation: Detected at run time if case-selectors checking was enabled during compilation.

In a FOR statement, the type of the initial value is not assignment compatible with the type of the control variable, and the statement in the loop body is executed.

Explanation: Detected at run time if subrange checking was enabled during compilation. Assignment compatiblity is not enforced if the statement in the loop body can never be executed.

In a FOR statement, the type of the final value is not assignment compatible with the type of the control variable and the statement in the loop body is executed.

Explanation: Detected at run time if subrange checking was enabled during compilation. Assignment compatibility is not enforced if the statement in the loop body can never be executed.

When an integer is being read from a text file, the digits read do not constitute a valid integer value. (Initial spaces and end-of-line markers are skipped.)

Explanation: Detected at run time.

When an integer is being read from a text file, the type of the value read is not assignment compatible with the type of the variable.

Explanation: Detected at run time if subrange checking was enabled during compilation.

When a real number is read from a text file, the digits read do not constitute a valid real number. (Initial spaces and end-of-line markers are skipped.)

Explanation: Detected at run time.

The value of the file buffer variable is undefined when a READ or READLN procedure is performed.

Explanation: Not detected.

A WRITE or WRITELN procedure specifies a field width in which the integers representing the total width and the number of fractional digits are less than 1.

Explanation: Not detected.

The bounds of an array passed to a conformant array parameter are outside the range specified by the conformant array's index type.

Explanation: Detected at run time if bounds checking was enabled during compilation.

Appendix G
Description of Implementation Features

The ISO standard for PASCAL allows some features of the language to be defined by a particular implementation or dependent on an implementation. This appendix lists all features of VAX–11 PASCAL that are implementation-defined or implementation-dependent, and explains how these features are treated by VAX–11 PASCAL.

G.1 Implementation-Defined Features

The value of each character allowed in a character string

VAX–11 PASCAL Treatment: See Appendix A.

The range of real number values represented by the type REAL

VAX–11 PASCAL Treatment: See the *VAX Architecture Handbook*.

The characters represented by the type CHAR and their ordinal values

VAX–11 PASCAL Treatment: See Appendix A.

The point at which the REWRITE, PUT, RESET, and GET procedures are performed on a file

VAX–11 PASCAL Treatment: Performed immediately unless the file is a terminal file, in which case delayed device access occurs (see Section 8.10)

The value of MAXINT

VAX–11 PASCAL Treatment: 2,147,483,647

The accuracy to which the results of real-number operations are calculated

VAX–11 PASCAL Treatment: See the *VAX Architecture Handbook* and the *VAX–11 Record Management Services Reference Manual.*

Default field widths

VAX-11 PASCAL Treatment:

Values of type INTEGER	10
Values of type REAL	12
Values of type BOOLEAN	6

The number of digits used to represent the exponent of a floating-point number

VAX-11 PASCAL Treatment:

F_floating or D_floating	2
G_floating	3
H_floating	4

The value of the exponent character

VAX-11 PASCAL Treatment: 'E'

The case (upper or lower) in which the Boolean values TRUE and FALSE are printed as output

VAX-11 PASCAL Treatment: Uppercase; that is, TRUE and FALSE

The effect of the PAGE procedure

VAX-11 PASCAL Treatment: PAGE writes a line containing only the form-feed character (ASCII value 12)

The binding of a file variable whose name is listed in the program heading

VAX-11 PASCAL Treatment: The file name (unless it is INPUT or OUTPUT) is equated to a logical name, if a translation for the file name exists. If there is no corresponding translation, the file type DAT is appended to the name listed in the heading, as in INFILE.DAT. If the file name is INPUT, the file is equated to PAS$INPUT, if PAS$INPUT is defined; otherwise, the file is equated to SYS$INPUT. Similarly, if the file name is OUTPUT, the file is equated to PAS$OUTPUT, if PAS$OUTPUT is defined; otherwise, the file is equated to SYS$OUTPUT.

G.2 Implementation-Dependent Features

The order of evaluation of the following items:

- Index values of an array variable
- Expressions in a set constructor
- Operands in a dyadic operation

VAX-11 PASCAL Treatment: Random order

Order of evaluation, accessing, and binding of actual parameters in a function designator and a procedure call

VAX-11 PASCAL Treatment: Random order

Order of accessing the variable and evaluating the expression in an assignment statement

VAX-11 PASCAL Treatment: Random order

The effect of reading a text file for which the PAGE procedure was called

VAX-11 PASCAL Treatment: Page reads a line containing only the form-feed character (ASCII value 12).

The binding of a file variable whose name is listed in the program heading to entities that are external to the program

VAX-11 PASCAL Treatment: Reported as an error at compile time

Appendix H
Program Examples

This appendix contains four programs that perform the following tasks:

- Program 1 adds, deletes, and updates records in an indexed file and produces a list of records sorted by customer number, last name, and zip code.
- Program 2 reads hexadecimal input typed at the terminal and converts it to decimal. This program simulates the behavior of VAX-11 Run-Time Library input procedures.
- Program 3 writes a message in reverse video to the terminal screen, then prompts for and accepts information typed after the message.
- Program 4 counts the number of occurrences of each word in a file and prints an alphabetized list of the words.

H.1 Update Indexed File

```
PROGRAM Update_File (OUTPUT, Addresses, Transactions, SortOut);

TYPE
   Code = (A, D, C);
   String = VARYING [35] OF CHAR;
   (* Record of customer information *)
   Address_Record = RECORD
                    Customer_No : [KEY(0)] PACKED ARRAY [1..8] OF CHAR;
                    Last_Name : [KEY(1)] PACKED ARRAY [1..25] OF CHAR;
                    First_Name : String;
                    Initial : CHAR;
                    Address : String;
                    City : String;
                    State : PACKED ARRAY [1..2] OF CHAR;
                    Zip : [KEY(2)] PACKED ARRAY [1..5] OF CHAR;
                    END;

VAR
   (* Master file of customers *)
   Addresses : FILE OF Address_Record;

   (* Input file of transactions *)
   Transactions : TEXT;
   Tcode : Code;
   Customer_No : PACKED ARRAY [1..8] OF CHAR;
   Last_Name : PACKED ARRAY [1..25] OF CHAR;
   Initial : CHAR;
```

```
   First_Name, Address, City : String;
   State : PACKED ARRAY [1..2] OF CHAR;
   Zip : PACKED ARRAY [1..5] OF CHAR;
   Record_Number : INTEGER := 1;
   (* Sorted output file *)
   SortOut : TEXT;
   I : INTEGER;

PROCEDURE Add_Record
   (Rec_Num : INTEGER);

   (* This procedure adds a record to the master file Addresses *)

   VAR
      Arec : Address_Record;

   BEGIN
   READLN (Transactions, Arec.Customer_No);
   FINDK (Addresses, 0, Arec.Customer_No);
   IF UFB (Addresses)
   THEN
      BEGIN
         WITH Arec DO
            BEGIN
            READLN (Transactions, Last_Name);
            READLN (Transactions, First_Name);
            READLN (Transactions, Initial);
            READLN (Transactions, Address);
            READLN (Transactions, City);
            READLN (Transactions, State);
            READLN (Transactions, Zip);
            WRITE (Addresses, Arec);
            END
      END
   ELSE
      BEGIN
      WRITELN ('File already contains record for ', Arec.Customer_No);
      WRITELN ('New record ', Rec_Num:3, ' not added');
      FOR I := 1 TO 7 DO
         READLN (Transactions);
      END;
   END;

PROCEDURE Delete_Record
   (Rec_Num : INTEGER);

   (* This procedure deletes a record from the master file Addresses *)

   BEGIN
   READLN (Transactions, Customer_No);
   FINDK (Addresses, 0, Customer_No);
   IF NOT UFB (Addresses)
   THEN
      DELETE (Addresses)
   ELSE
      BEGIN
      WRITELN ('File does not contain record for ', Customer_No);
      WRITELN ('Record number ', Rec_Num:3, ' not deleted');
      END;
    END;

PROCEDURE Change_Record
   (Rec_Num : INTEGER);

   (* This procedure updates a record in the master file Addresses
      with the new information provided in the Transactions file *)
```

```
    BEGIN
    READLN (Transactions, Customer_No);
    FINDK (Addresses, 0, Customer_No);
    IF NOT UFB (Addresses)
    THEN
       BEGIN
       READLN (Transactions, Addresses^.Address);
       READLN (Transactions, Addresses^.City);
       READLN (Transactions, Addresses^.State);
       READLN (Transactions, Addresses^.Zip);
       UPDATE (Addresses);
       END
    ELSE
       BEGIN
       WRITELN ('File does not contain record for ', Customer_No);
       WRITELN ('Record number ', Rec_Num:3, ' not changed');
       FOR I := 1 TO 4 DO
          READLN (Transactions);
       END;
    END;

PROCEDURE Number_Sort;

    (* This procedure produces a list of customers sorted by number *)

    VAR
       Format_String : [STATIC] VARYING [10] OF CHAR := '';

    BEGIN
    RESETK (Addresses, 0);
    OPEN (SortOut);
    REWRITE (SortOut);
    WRITELN (SortOut, 'Customer Number':17, 'Last Name':19,
          'Zip Code':28);
    WRITELN (SortOut);
    WRITELN (SortOut);
    WHILE NOT EOF (Addresses) DO
       BEGIN
       WRITELN (SortOut, Format_String:5, Addresses^.Customer_No,
          Format_String:10, Addresses^.Last_Name, Format_String:10,
          Addresses^.Zip);
       GET (Addresses);
       END;
    END;

PROCEDURE Name_Sort;

    (* This procedure produces a list of customers sorted by last name *)

    VAR
       Format_String : [STATIC] VARYING [10] OF CHAR := '';

    BEGIN
    RESETK (Addresses, 1);
    PAGE (SortOut);
    WRITELN (SortOut, 'Last Name':19, 'Customer Number':33,
          'Zip Code':13);
    WRITELN (SortOut);
    WRITELN (SortOut);
    WHILE NOT EOF (Addresses) DO
       BEGIN
       WRITELN (SortOut, Format_String:5, Addresses^.Last_Name,
          Format_String:10, Addresses^.Customer_No, Format_String:10,
          Addresses^.Zip);
       GET (Addresses);
       END;
    END;
```

```
PROCEDURE Zip_Sort;

   (* This procedure produces a list of customers
      sorted by zip code *)

   VAR
      Format_String : [STATIC] VARYING [10] OF CHAR := '';

   BEGIN
   RESETK (Addresses, 2);
   PAGE (SortOut);
   WRITELN (SortOut, 'Zip Code':12, 'Last Name':22,
         'Customer Number':33);
   WRITELN (SortOut);
   WRITELN (SortOut);
   WHILE NOT EOF (Addresses) DO
      BEGIN
      WRITELN (SortOut, Format_String:5, Addresses^.Zip, Format_String:10,
         Addresses^.Last_Name, Format_String:10, Addresses^.Customer_No);
      GET (Addresses);
      END;
   CLOSE (SortOut);
   END;

BEGIN
OPEN (Addresses,
      HISTORY := UNKNOWN,
      ORGANIZATION := INDEXED,
      ACCESS_METHOD := KEYED);
OPEN (Transactions,
      'DISK$WORK:[RECORDS]TRANS.DAT',
      HISTORY := OLD);
RESETK (Addresses, 0);
RESET (Transactions);
WHILE NOT EOF (Transactions) DO
   BEGIN
   READLN (Transactions, Tcode);
   (* Determine whether record is to be added, deleted, or changed
      and call the appropriate procedure to process it *)
   CASE Tcode OF
      A : Add_Record (Record_Number);
      D : Delete_Record (Record_Number);
      C : Change_Record (Record_Number);
   END;
   Record_Number := Record_Number + 1;
   END;
(* Produce sorted output file *)
Number_Sort;
Name_Sort;
Zip_Sort;
CLOSE (Addresses);
CLOSE (Transactions);
END.
```

H.2 Hexadecimal Input

```
PROGRAM Read_Hex (INPUT, OUTPUT);

LABEL
    1,          { Value successfully converted }
    2,          { End of white space }
    99;         { Error, flush remainder of line }

CONST
    Prompt = 'Enter a hex number: ';
    Space_or_tab = [' ', '	'];
    Hex_Digits = ['0'..'9', 'A'..'F', 'a'..'f'];

VAR
    Hex_Value : INTEGER;

BEGIN
WRITELN;
WRITE (Prompt);
WHILE TRUE DO
   BEGIN
   IF EOLN (INPUT)
   THEN
      { No input on this line - put out a new prompt }
      BEGIN
      WRITE (Prompt);
      READLN;
      END;
   WHILE NOT EOLN (INPUT) DO
      { Skip leading white space }
      IF INPUT^ IN Space_or_tab
      THEN
         GET(INPUT)
      ELSE
         GOTO 2;
      2:
      IF NOT EOLN (INPUT)
      THEN
         { Not a blank line so hex value should follow }
         BEGIN
         IF NOT (INPUT^ IN Hex_Digits)
         THEN
            { First character that was not space or tab
              was not hex either - error }
            BEGIN
            WRITELN ('Illegal hex value');
            GOTO 99;
            END
         ELSE
            { At least one hex character }
            BEGIN
            Hex_Value := 0;
            REPEAT
               BEGIN
               IF NOT (INPUT^ IN Hex_Digits)
               THEN
                  { Next character is not a hex digit -
                    conversion complete }
                  GOTO 1
```

```
            ELSE
               { Check for overflow, then include this
                 digit in the accumulated value }
               BEGIN
               IF Hex_Value > %X'07FF FFFF'
               THEN
                  { Multiplying by 16 would cause value
                    to become negative }
                  BEGIN
                  WRITELN ('Hex value too large');
                  GOTO 99;
                  END
               ELSE
                  BEGIN
                  Hex_Value := Hex_Value * 16;
                  CASE INPUT^ OF
                     '0', '1', '2', '3', '4',
                       '5', '6', '7', '8', '9':
                        Hex_Value := Hex_Value - ORD ('0');
                     'A', 'B', 'C', 'D', 'E', 'F':
                        Hex_Value := Hex_Value - ORD ('A') + 10;
                     'a', 'b', 'c', 'd', 'e', 'f':
                        Hex_Value := Hex_Value - ORD ('a') + 10;
                  END;
                  Hex_Value := Hex_Value + ORD (INPUT^);
                  GET(INPUT);
                  END;
               END;
            END;
         UNTIL EOLN (INPUT);
         { End of line - conversion complete }
          GOTO 1;
        END;
      END;

    99:
    { Error previously reported - flush remainder of line }
    WHILE NOT EOLN (INPUT) DO
       GET (INPUT);
    END;

1:
WRITELN ('Value in decimal: ', Hex_Value);
END.
```

H.3 Screen Display

```
PROGRAM Screen_Routines (INPUT, OUTPUT);

(*   This program illustrates the use of the Run-Time     *)
(*   Library screen package routines.                     *)

TYPE
   Word_integer = [WORD] 0..65535;
   Form_line = VARYING [15] OF CHAR;
   Data_line = VARYING [30] OF CHAR;

VAR
   Screen_stat, Line, Column, Counter : INTEGER;
   Welcome_Msg : VARYING [50] OF CHAR;
   Form : ARRAY [1..3] OF Form_line;
   Userdata : ARRAY [1..3] OF Data_line;
   Reverse : Word_integer := 2; (* flags for LIB$PUT_SCREEN *)

(*******  Declare external RTL routines  *******)
```

```
[EXTERNAL] FUNCTION LIB$ERASE_PAGE
   (Line_no : Word_integer;
   Col_no : Word_integer)
   : INTEGER;
   EXTERN;

[EXTERNAL] FUNCTION LIB$PUT_SCREEN
   (Out_text : VARYING [C] OF CHAR;
   Line_no : Word_integer;
   Col_no : Word_integer;
   Flags : Word_integer := %IMMED 0)
   : INTEGER;
   EXTERN;

[EXTERNAL] FUNCTION LIB$SET_CURSOR
   (Line_no : Word_integer;
   Col_no : Word_integer)
   : INTEGER;
   EXTERN;

[EXTERNAL] FUNCTION LIB$GET_SCREEN
   (VAR Input_text : VARYING [U] OF CHAR;
   Prompt_str : VARYING [V] OF CHAR := %IMMED 0;
   VAR Out_len : Word_integer := %IMMED 0)
   : INTEGER;
   EXTERN;

[EXTERNAL] PROCEDURE LIB$STOP
   (%IMMED Cond_value : INTEGER);
   EXTERN;

(*******  Begin main program  *******)

BEGIN
Welcome_Msg := 'WELCOME! Please input data as requested. ';
Form[1] := 'Name:      ';
Form[2] := 'Address:   ';
Form[3] := 'Phone:     ';
(* Clear the entire screen *)
Screen_stat := LIB$ERASE_PAGE (Line_no := 1, Col_no := 1);
IF NOT ODD (Screen_stat)
THEN
   LIB$STOP (Screen_stat);
(* Write a welcome message to terminal in reverse video *)
Line := 3;
Column := 5;
Screen_stat := LIB$PUT_SCREEN (Welcome_Msg, Line, Column, Reverse);
IF NOT ODD (Screen_stat)
THEN
   LIB$STOP (Screen_stat);
(* Output prompts and collect data *)
Line := 5;
Column := 10;
FOR Counter := 1 TO 3 DO
   BEGIN
   Screen_stat := LIB$SET_CURSOR (Line_no := Line+Counter,
   Col_no := Column);
   IF NOT ODD (Screen_stat)
   THEN
      LIB$STOP (Screen_stat);
   Screen_stat := LIB$GET_SCREEN (Userdata[Counter], Form[Counter]);
   IF NOT ODD (Screen_stat)
   THEN
      LIB$STOP (Screen_stat);
   END;
END.
```

H.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;
   (* Record to define 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;

(* This function allocates storage and assigns an address *)

FUNCTION Create_Node : Ref_Tree_Node;

   VAR
      New_Node : Ref_Tree_Node;

   BEGIN
   NEW (New_Node);
   (* Initialize 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 in the tree. *)

PROCEDURE Enter_Node;

   VAR
      Current : Ref_Tree_Node;

   BEGIN
   Current := Root;
   (* Initialize 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 e 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 nonalphabetics, makes
   lowercase letters out of uppercase letters, and enters
   the word into the tree by calling Enter_Node *)
```

```
PROCEDURE Scan_File;

   VAR
      I : INTEGER;
      C : CHAR;

   BEGIN
   I := 0;
   C := ' ';
   (* Check for nonalphabetics *)
   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
         BEGIN
         (* Convert uppercase letters to lowercase *)
         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;
      (* Enter the word into the tree via the procedure *)
      IF I > 0
      THEN
         BEGIN
         FOR I e I + 1 TO Word_Length DO
            New_Word[I] := ' ';
            Enter_Node;
            I := 0;
         END;
      END;
   END;

(* Main program *)
BEGIN
Initialize;
Scan_File;
Print_Node (Root);
END
```

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

VAX-11 PASCAL
Language Reference
Manual
AA-H484C-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

BAR Code

- — — Do Not Tear - Fold Here and Tape — — — — 00 98648 — — — — — — — — — — —

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

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

— — Do Not Tear - Fold Here —