CRAY RESEARCH, INC.

CRAY COMPUTER SYSTEMS

PASCAL REFERENCE MANUAL

SR-0060

RECORD OF REVISION

CRAY RESEARCH, INC.

PUBLICATION NUMBER SR-0060

Each time this manual is revised and reprinted, all changes issued against the previous version are incorporated into the new version and the new version is assigned an alphabetic level.

Every page changed by a reprint with revision has the revision level in the lower righthand corner. Changes to part of a page are noted by a change bar in the margin directly opposite the change. A change bar in the margin opposite the page number indicates that the entire page is new. If the manual is rewritten, the revision level changes but the manual does not contain change bars.

Requests for copies of Cray Research, Inc. publications should be directed to the Distribution Center and comments about these publications should be directed to:

CRAY RESEARCH, INC.
2520 Pilot Knob Road
Suite 310
Mendota Heights, Minnesota 55120

Revision	Description
	September 1983 - Original printing.
A	September 1984 - This reprint with revision updates the manual for Pascal release 2.0. It adds the VALUE definition; the VIEWING statement; the IMPORTED, EXPORTED, COMMON, and STATIC variable declarations; the LOC function under pointer types; the SIZEOF function under DISPOSE; new compiler options; and new messages. Material from appendix F in the initial release is now incorporated in subsection 2.3. Miscellaneous technical and editorial corrections were also made. This manual obsoletes all previous printings.
B	January 1986 - This reprint with revision updates the manual for Pascal release 3.0. Release 3.0 adds a set of array processing constructs; automatic vetorization of FOR loops that comply with Pascal vectorization rules; CPU targeting; additional debugging and listing options; constant expressions in constant definitions; the sharing of data in FORTRAN TASK COMMON blocks; the I32 data type; conditional expressions; and various changes for running under the Cray operating system UNICOS†. Changes have been made to the Pascal syntax to accommodate the changes and additions for this release. This reprint also includes miscellaneous technical and editorial corrections. This manual obsoletes all previous printings.

† UNICOS is derived from the AT&T UNIX system; UNIX is a trademark of AT&T Bell Laboratories.

PREFACE

This publication is a reference manual for the Pascal programming language as implemented by Cray Research, Inc. (CRI). It is not a tutorial, although it does illustrate the use of Pascal with frequent examples.

Many books that teach the language are available. The following publications are frequently used descriptions of Pascal:

Cooper, Douglas. *Standard Pascal User Reference Manual*. New York: W. W. Norton.

Grogono, Peter. *Programming in Pascal*. Addison-Wesley, 1978.

Jensen, Kathleen and Niklaus Wirth. *Pascal User Manual and Report*. 3rd ed. New York: Springer-Verlag, 1985.

Wilson, Ian and Tony Addyman. *A Practical Introduction to Pascal*. MacMillan and Springer-Verlag, 1978.

The book by Wilson and Addyman includes a complete description of the International Standards Organization (ISO) Pascal standard, ISO/DIS 7185. The following CRI publications may also be helpful to the Pascal programmer:

SR-0011	COS Version 1 Reference Manual
SR-0012	Macros and Opdefs Reference Manual
SD-0061	Pascal Internal Reference Manual
SR-0113	Programmer's Library Reference Manual
SM-0114	System Library Reference Manual
SR-2011	UNICOS Commands Reference Manual
SR-2013	CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual
SR-2014	UNICOS File Formats and Special Files Reference Manual

CONTENTS

FIGURES

TABLES

INDEX

1. INTRODUCTION

Pascal is a high level, general purpose computer language designed in 1970 by Professor Niklaus Wirth of the Federal Institute of Technology at Zurich, Switzerland. The language emphasizes the virtues of structure, simplicity, and portability. Using Pascal, you can implement algorithms and data structures in a high-level, machine-independent manner without sacrificing efficiency.

The Cray Pascal Compiler transforms Pascal code into machine language instructions that execute on CRAY-2, CRAY X-MP, and CRAY-1 Computer Systems. The compiler and programs created by it run under the Cray operating systems COS and UNICOS.

Cray Pascal complies with the Level 1 requirements of standard ISO/DIS 7185, defined by the International Standards Organization (ISO), with three restrictions. The restrictions, as well as Cray Research Inc. (CRI) extensions to the standard, are described later in this section. This manual notes all CRI extensions.

Section 2 describes the Pascal invocation statement, which tells the operating system under which Pascal is operating to load and execute the compiler. The compiler can be invoked with the UNICOS command line or the COS control statement. The UNICOS Commands Reference Manual describes UNICOS and its system commands in more detail. The COS Version 1 Reference Manual describes COS and its control statements in more detail.

Sections 3 through 13 describe both the standard features of Pascal and the CRI enhancements.

Section 3 provides general information on the language, including its notation, vocabulary, and format. Commenting is also described.

Section 4 explains the organization of the program and defines the syntax for the PROGRAM heading.

Section 5 describes Pascal data types and the statements associated with them.

Section 6 describes the constructs that are available for array processing.

Section 7 defines the WITH and VIEWING statements.

Section 8 explains program control statements, including the simple assignment statement and more complex compound statements.

Section 9 describes procedures and functions, including how they are invoked and how parameters are passed.

Section 10 explains I/O features available with Pascal.

Section 11 describes dynamic allocation and the associated NEW and DISPOSE statements.

Section 12 describes compile units called modules that can be maintained as library routines and called from Pascal programs or other modules.

Section 13 describes automatic vectorization of FOR loops and the optimization of code by the use of specific parameters on the Pascal invocation statement.

Appendixes give the ASCII character set, describe the predefined procedures and functions available, list both compiler and execution error messages, offer a summary of the syntax for Pascal statements, describe debugging aids, list errors not detected by the compiler, and give examples of I/O features.

1.1 EXTENSIONS

The CRI extensions to the ISO Level 1 Pascal standard include the following:

- ALFA is a predefined type specified as an eight-member packed array of characters.
- Array processing features allow operations on entire arrays as follows:
  - Array expressions
  - Array merges
  - Array relational operators
  - Array valued field and pointer accesses
  - Slice index specifications
- BY clause allows you to specify an increment index with FOR loops.
- CACHE† declarations allow definition of common blocks residing in Local Memory

† Available with CRAY-2 Computer Systems only

- COMMON declarations allow FORTRAN common blocks to share data across compile units.

- Conditional expressions can be used on the right-hand side of an assignment statement.

- Constant declarations allow constant expressions.

- $DEBUG is a predefined constant.

- EXTERNAL and FORTRAN procedure and function directives allow using routines outside of a program.

- I24 is a predefined 24-bit integer data type.

- I32 is a predefined 32-bit integer data type.

- IMPORTED and EXPORTED procedures permit the use of previously compiled modules and FORTRAN or Cray assembly language (CAL) subroutines.

- IMPORTED and EXPORTED declarations allow sharing of variables between compile units.

- MODULE compile units allow you to define Pascal procedures, functions, and data without a PROGRAM module.

- Integers can be expressed in octal notation

- OTHERWISE label specifies the action to take when no other label in a CASE statement is selected.

- Identifiers can include special characters ($ % @ _).

- STATIC declarations allow a local variable to keep its value between calls to a routine.

- TASKVAR† declarations allow TASK COMMON blocks to be specified.

- The predefined procedure and function list is expanded to include the following:

  - ARCSIN and ARCCOS calculate the inverse of the sine and cosine.

  - BAND, BOR, BXOR, and BNOT are bit-string Boolean functions that accept integer arguments and return integers.

† Not available on CRAY-2 Computer Systems

- CONNECT associates a COS local dataset or a UNICOS file with a Pascal file.
- HALT terminates the execution of a program when encountered.
- LOC returns the address of a variable.
- LOG calculates the common logarithms.
- Reduction functions (ANY, ALL, MAXVAL, MINVAL, PRODUCT, and SUM) reduce array expressions into simple values.
- LSHIFT and RSHIFT shift an integer argument left or right by a specified number of places.
- SINH, COSH, and TANH are the hyperbolic functions.
- SIZEOF gives the size of a dynamic variable.
- TAN calculates a tangent.

- VALUE definitions initialize variables at compile time.
- VIEWING statement allows a variable to be used to see different types.

1.2 RESTRICTIONS

Cray Pascal includes the following restrictions to the ISO Level 1 Pascal standard.

- The commercial at sign (@) cannot be used as a substitute character for the circumflex (^), as specified by the standard. The @ is implemented as a valid character in an identifier (see section 3, Pascal Vocabulary).
- The maximum line length for a text file is 140 characters. The standard does not restrict the line length of a text file.
- Some errors specified by the standard are not detected. Appendix G, Errors not Reported by Cray Pascal, details these errors.

1.3 CONVENTIONS

This manual observes the Backus-Naur Form (BNF) conventions in representing the language's syntax. Any word not enclosed in quotation marks is called a *nonterminal symbol*. Each nonterminal symbol in a BNF construction is defined in turn; however, BNF descriptions are not expanded fully in the text. For example, the terminal symbols for letter and digit are not defined every time an identifier is required in the syntax. See appendix E, Pascal Syntax, for a complete description of Pascal syntax.

The following notation is used:

Symbol	Description
x|y	Indicates that either x or y is valid
"x"	Indicates that x is a literal, or terminal, element to be entered exactly as specified. The terminal elements include reserved words and predefined identifiers, such as PROGRAM in the program heading. (The reserved words and predefined identifiers are listed in section 3, Pascal Vocabulary.)
[x]	Indicates 0 or 1 occurrence of x is valid
{x}	Indicates 0 or more occurrences of x are valid
()	Indicates a grouping. Parentheses have no syntactic significance of their own. For example: `id = (letter|"$"\|"%"\|"@") { letter\|digit\|"_"\|"$"|"%"|"@" } .`
.	Indicates the end of a description

In sample programs and programming examples, uppercase indicates reserved words and predefined identifiers, and lowercase indicates identifiers chosen by you.

This manual uses the conventional Pascal definition rather than the COS definition of the word *file*. A Pascal file is the same thing as a COS blocked dataset or a standard UNICOS file. Pascal does not support multi-file datasets.

2. USING PASCAL ON A CRAY COMPUTER

The Pascal compiler runs under both Cray operating systems, COS and UNICOS.

The source file of Pascal statements remains virtually the same regardless of the operating system. The main differences between using Pascal under COS and under UNICOS center around how the compiler is invoked and how the job is executed, since COS is primarily a batch operating system and UNICOS is primarily interactive.

2.1 USING PASCAL UNDER COS

A typical Pascal job on the COS operating system contains the following files:

- A job control language (JCL) file of COS control statements
- A source file of Pascal statements
- One or more data files (optional)

End-of-file (EOF) indicators divide files from each other. An end-of-dataset (EOD) indicator follows the last file. (The actual representation of the end-of-file and end-of-dataset indicators depends on the front-end computer.) Together these files comprise a job dataset named $IN by COS. Figure 2-1 shows a COS job dataset with one data file in card deck format.

The dataset containing the job is typically submitted to the Cray computer system for processing through a front-end computer. (The method of submitting a job depends on the front-end computer.)

A job's output dataset (named $OUT by default) is returned to the front-end computer when the job completes. The job's output dataset includes a program listing (by default), any output created by the job that is written to file OUTPUT†, and the job's logfile. The COS Version 1 Reference Manual describes the logfile, which contains a history of the job and other aspects of running a job on COS.

† Output written to any other file is not automatically returned.

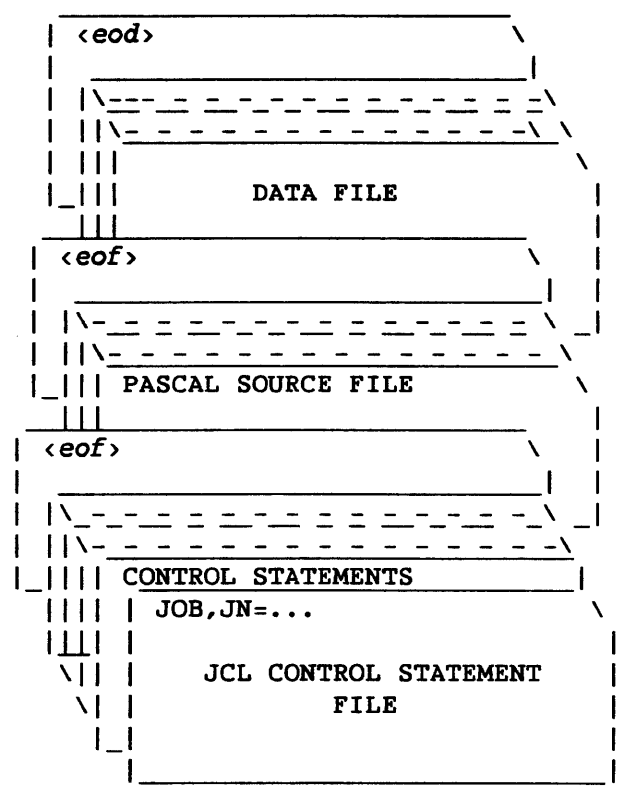

Figure 2-1. A Typical Job Dataset

2.1.1 JCL FILE

A simple Pascal job may contain the following COS control statements in its JCL file:

```
JOB,JN=TEST.
ACCOUNT,AC= ... .
PASCAL.
SEGLDR,GO.
/EOF
```

The JOB statement is a required statement that defines the job to COS. At the minimum, it must contain a JN parameter to assign the job a name.

The ACCOUNT control statement presents the user's account number, which may be required by a site before access is granted to the system.

The PASCAL statement causes the Pascal compiler to be loaded and executed. Since no input dataset is specified as a parameter in this example, the compiler looks for Pascal source statements following the end-of-file indicator, /EOF. See the following subsection for a description of the parameters available on the PASCAL control statement. The result of the compilation in this example is an executable binary program.

The SEGLDR statement with the GO parameter loads and executes the binary program. It also automatically accesses the Pascal run-time library, $PSCLIB.

The COS Version 1 Reference Manual describes these and other control statements.

2.1.2 COS PASCAL CONTROL STATEMENT

The PASCAL control statement invokes the Pascal compiler under COS. You select compiler parameters either explicitly by listing them on the control statement or implicitly by accepting the default values. All parameters are optional and have default values. The format of the PASCAL control statement is as follows:

```
PASCAL[,I=idn][,L=ldn][,B=bdn][,O=list][,CPU=list].
```

I=*idn* Specifies the dataset containing the Pascal source code. *idn* is the name by which the source code is referenced in COS. The default is $IN.

L=*ldn* Specifies the dataset to receive the job's list output. The default for *ldn* is $OUT. If L=0 is specified, all list output except fatal error messages is suppressed. Fatal error messages are written to $OUT if L=0.

B=*bdn* Specifies the dataset to receive the binary load modules generated by the compiler. *bdn* is the name by which the binary load modules are referenced in COS. The default is $BLD.

O=*list* Specifies compiler options, separated by colons, in effect at the beginning of the compilation. Compiler directives placed inside comments in the Pascal program override the initial settings. For more information about compiler directives, see Compiler Directives later in this section. As listed in table 2-1, defaults are as follows:

```
A-:BP-:BREG=8:BT-:C-:D+:DEBUG-:DM0:H2:H+24:L+:O+^
:P-:P24:R+:RV-:S4:S+4:ST-:T+:TREG=8:U-:V+:X-:Z+
```

CPU=*list* Specifies the characteristics of the Cray Computer System for which Pascal is to generate code. The format of the CPU command is as follows:

CPU = [*primary*] { ":"*characteristic* }

primary can be one of the following:

Target Machine	Description
CRAY-X4	Generates code for a four-processor CRAY X-MP
CRAY-X2	Generates code for a dual-processor CRAY X-MP
CRAY-X1	Generates code for a single-processor CRAY X-MP
CRAY-XMP	Generates code for a single-processor CRAY X-MP that also runs on four-processor and dual processor CRAY X-MPs
CRAY-1M	Generates code for a CRAY-1 Model M
CRAY-1S	Generates code for a CRAY-1 Model S
CRAY-1B	Generates code for a CRAY-1 Model B
CRAY-1A	Generates code for a CRAY-1 Model A
CRAY-1	Generates code for a CRAY-1 Model A that also runs on CRAY-1 Models B, S, and M

characteristic can be one of the following traits:

Trait	Description
EMA	Causes Pascal to generate 24-bit A register immediate load instructions, where necessary, and allows the use of common blocks larger than 4 million words
NOEMA	Disables the generation of 24-bit A register immediate load instructions and disallows the use of common blocks larger than 4 million words
CIGS	Enables compressed index and gather/scatter
NOCIGS	Disables compressed index and gather/scatter
VPOP	Enables vector population and parity
NOVPOP	Disables vector population and parity
READVL	Enables vector length read instructions
NOREADVL	Disables vector length read instructions
MEMSIZE	The format of the MEMSIZE option is as follows: MEMSIZE = *n* ["K" | "M"] . MEMSIZE is *n* * 1024 words for *n*K and *n* * 1048576 words for *n*M.
BDM	Enables bidirectional memory
NOBDM	Disables bidirectional memory

The CPU option cannot be specified with a CRAY-2 Computer System. For more information about the CPU parameter, see CPU Targeting later in this section.

2.2 USING PASCAL UNDER UNICOS

Since UNICOS is an interactive operating system, you need not include a JCL file in with the Pascal source statements as is the case with COS. In fact, UNICOS does not support multifile datasets. The source statements and the data are each in separate files. Instead of the JCL file, you enter the commands necessary to compile, load, and execute your program as UNICOS commands at your terminal.

The following commands show how to run a Pascal program on UNICOS. (The $ is the default UNICOS prompt.)

```
$ pascal -i test.p
$ ld a.o
$ a.out <test.data
```

The first command invokes the Pascal compiler, which compiles the source statements found in the file test.p and generates an executable file with the default file name a.o. If test.p is not in the current directory, you can specify a path name.

The ld command loads the a.o file from the compilation and names it a.out. The a.out command, with the data file test.data as input, executes the binary program.

2.2.1 UNICOS PASCAL COMMAND LINE

The pascal command line invokes the Pascal compiler under UNICOS. You select compiler parameters either explicitly by listing them on the command line or implicitly by accepting the default values. All parameters are optional and have default values. The format of the pascal command line is as follows:

```
pascal -i idn -l ldn -b bdn -o list
```

-i *idn* Specifies the file containing the Pascal source code. When *idn* is not a complete path name, the input file defaults to a working directory. The default is stdin.

-l *ldn* Specifies the file receiving the job's list output. If -l 0 is specified, all list output is suppressed. The default is stdout.

-b *bdn* Specifies the file receiving the binary load modules generated by the compiler; the default is a.o.

-o *list* Specifies compiler options, separated by commas, in effect at the beginning of the compilation. The compiler options available under UNICOS are generally the same as the compiler options available under COS. Table 2-1 notes differences among CRAY-2, CRAY X-MP, and CRAY-1 Computer Systems and between COS and UNICOS.

Compiler directives placed inside comments in the Pascal program override the initial settings. The defaults on a CRAY-1 or a CRAY X-MP Computer System running UNICOS are as follows:

```
A-,BP-,BREG=8,BT-,C-,D+,DM0,L+,O+,P-,
P24,R+,RV+,ST-,T+,TREG=8,U-,V+,X-,Z+
```

The defaults on a CRAY-2 Computer System are as follows:

```
A-,BP-,BREG=8,BT-,C-,D+,DM0,L+,O+,P-,
P32,R+,RV+,ST-,T+,TREG=8,U-,V+,X-,Z+
```

2.3 COMPILER DIRECTIVES

Directives are issued to the Pascal compiler in two ways:

- As arguments to the O (options) parameter on the PASCAL control statement
- Within comments in the Pascal source code (restricted for some directives; see introduction to table 2-1)

The format of a comment directive is as follows:

```
comment-directive =
   "(*#" directive { ":" directive } [ " " comment ] "*)"
```

Text following the directive and preceding the symbol that closes the comment is treated as a comment.

Examples of formats:

1. In the PASCAL control statement: PASCAL,O = X+:L+:BREG = 14.
2. In a comment: (*#X+:L+:BREG=14 *)

Table 2-1 describes all compiler options. In general, an option followed by a plus sign enables that option, and an option followed by a minus sign disables that option. Some directives included in source comments are ignored if they appear after the keywords PROGRAM and MODULE. Directives and comments following the final period in a compile unit are attached to the next program or module, if any. Table 2-1 indicates directives that cannot be used within a program or module by an asterisk (*) in the column labeled Res (restricted).

Table 2-1. Compiler Options

Option	Res	Default	Action	
A		A-	A+	Aborts the job at the end of the compile step if compilation errors are found
			A-	Continues the job even if errors are encountered in the compilation
BP		BP-	BP+	Enables breakpoint checking by calling run-time library routine P$DBP before every statement. (See appendix F, Debug Information.)
			BP-	Disables breakpoint checking
BREG	*	BREG=8	BREG=*n*	Allocates *n* B registers to the first *n* user variables declared with VAR in a program or module. Variables that can be allocated to B registers include pointers, I24, CHAR, and any other simple type that it can fit in a 24-bit B register, but not Boolean. All available B registers not allocated to program- or module-level variables are reserved for use by variables at innerscopes. The limit to *n* depends on the number of procedures and parameters; the compiler grants as much of a request as is available. BREG=0 implies that all B registers are reserved for nonglobal use. CRAY-2 Computer Systems use B and T registers by mapping into local common block LOCAL@CB. Up to 128 B and T registers can be specified. The default is eight global B and T registers with the rest of the block designated as nonglobal.
			BREG-	Uses no B registers for user variables
BT		BT-	BT+	Generates calls to FLOWTRACE routines to get a FLOWTRACE listing at the normal program exit
			BT-	Does not generate calls to FLOWTRACE routines

Table 2-1. Compiler Options (continued)

Option	Res	Default	Action	
C	*	C-	C+	Enables pseudo-CAL listings
			C-	Disables pseudo-CAL listings
D	*	D+	D+	Generates data for a dump if the program aborts during execution
			D-	Does not generate data for a dump
DEBUG	*	DEBUG-	DEBUG+	Controls the value of the predefined Boolean constant $DEBUG. $DEBUG=TRUE only when DEBUG+ is specified.
			DEBUG-	Specifies that $DEBUG=FALSE
DM	*	DM0	DM*n*	Sets the debug mode, which determines which code addresses are recorded in the Debug Symbol Table and how much optimization, vectorization, and scheduling can occur. Code addresses are used by the postmortem dump tools to identify where the program stopped. With *n*>1, scalar optimization and automatic vectorization are turned off, scheduling is limited (since there are more labels and scheduling only occurs between labels), and dead code and dead procedure elimination are suppressed.
			DM0	Suppresses the Debug Symbol Table
			DM1	Provides minimal debug support in the Debug Symbol Table: information about types, variables, constants, the use of nonlocal variables in procedures, and addresses of user-defined labels and procedure entry points.
			DM2†	Provides intermediate debug support in the Debug Symbol Table: everything included in minimal mode, plus addresses of compiler-generated labels in structured statements.

† Deferred implementation

Table 2-1. Compiler Options (continued)

Option	Res	Default	Action	
			DM3†	Provides full debug support in the Debug Symbol Table: everything included in intermediate mode, plus the address of each statement.
F		n/a	F	Forces a page eject in the listing. Cannot be used in the control statement.
H		H2	H*n*	Sets the requirements for heap (dynamic allocation) memory space in addition to the heap space used for the initial stack. *n* is specified in octal units of 1000 words. The default of H2 specifies 2000_8 (1024_{10}) words in the heap memory space.
H+*n*		H+24	H+*n*	Increases heap memory in octal units of 1000 words when the heap runs out of free space and must request additional memory from the operating system. The default of H+24 specifies that a minimum of 24000_8 (10240_{10}) words is requested for each heap expansion. If the heap increment is zero, a fatal error occurs when the heap runs out of free space. The H+*n* parameter cannot be specified with a CRAY-2 Computer System.
ISO	*	Off	ISO	Disables CRI extensions to the ISO Level 1 Pascal standard and issues error messages if extensions are encountered. If ISO is not specified (the default), extensions to the standard are enabled.
L		L+	L+ L-	Prints a source listing of the program Suppresses the source listing of the program. (If the L parameter on the PASCAL control statement is set to 0, O=L- is implied.)

† Deferred implementation

Table 2-1. Compiler Options (continued)

Option	Res	Default	Action	
O	*	O+	O+	Enables all optimizations
			O-	Disables all optimizations
			O*n*	Enables optimization with the tuning factor specified by *n* ($1 \leq n \leq 9$). High tuning factors increase compilation time but can improve the generated code.
P		P-	P+	Ejects a page before every routine so that no single page contains more than one routine
			P-	Does not eject a page before routines
P24	*	P24		Uses 24-bit pointers in packed structures; the default for CRAY-1 and CRAY X-MP Computer Systems.
P32	*	P32		Uses 32-bit pointers in packed structures; the default for CRAY-2 Computer Systems.
R		R+	R+	Generates code to do range checks
			R-	Does not generate code for range checks
				R+ is the same as RA+:RR+:RP+:RS+:RL+, and R- is the same as RA-:RR-:RP-:RS-:RL-.
RA		RA+	RA+	Generates code to check array indexes
			RA-	Does not generate array index checks
RL		RL+	RL+	Generates code to check shape compatibility (arrays must possess the same number of dimensions and corresponding dimensions must be the same size) in array expressions at run time.
			RL-	Does not generate code to check shape compatibility in array expressions
RP		RP+	RP+	Generates code to perform full runtime validity checks on pointer accesses
			RPN	Generates code to check pointers only for zero values at runtime

Table 2-1. Compiler Options (continued)

Option	Res	Default	Action	
			RP-	Does not generate code for pointer checks
RR		RR+	RR+	Generates subrange assignment checks
			RR-	Does not generate subrange assignment checks
RS		RS+	RS+	Generates code to check set use
			RS-	Does not generate set-checking code
RV		RV-	RV+	Generates code to perform variant checking
			RV-	Does not generate code for variant checking
R^		n/a	R^	Saves current settings of the R options for restoring with R* option. R^ cannot appear on the control statement.
R*		n/a	R*	Restores settings of the R options with those previously saved with the R^ option. R* cannot appear on the control statement.
S	*	S4	S*n*	Sets the requirements for stack memory space. *n* is specified in octal units of 1000 words. The default of S4 specifies 4000_8 (2048_{10}) words in the stack memory space. The S parameter cannot be specified with a CRAY-2 Computer System.
S+*n*		S+4	S+*n*	Increases stack memory in octal units of 1000 words. The default of S+4 specifies that when the stack becomes full, an additional 4000_8 (2048_{10}) words are added to the stack. If the stack increment is zero, a stack overflow causes a fatal error. The S+*n* parameter cannot be specified with a CRAY-2 Computer System.

Table 2-1. Compiler Options (continued)

Option	Res	Default	Action
ST		ST-	ST='*string*' Includes a subtitle line containing the first 72 characters of the specified string as the second line of the pageheader. The directive performs no character editing (blank suppression or conversion to uppercase). ST- Does not include a subtitle line in the page header
T		T+	T+ Generates code for stack overflow check T- Does not generate code for stack check. If stack overflow checking is disabled, the user stack does not expand and may overwrite the heap.
TREG	*	TREG=8	TREG=*n* Allocates *n* T registers to the first *n* user variables declared with VAR in a program or module. Variables that can be allocated to T registers include integers, Booleans, and any simple user-defined variable whose type indicates it can fit in a 64-bit T register. All available T registers that are not allocated to program or module level variables are reserved for use by variables at inner scopes. CRAY-2 Computer Systems use B and T registers by mapping into local common block LOCAL@CB. Up to 128 B and T registers can be specified. The default is eight global B and T registers with the rest of the block designated as nonglobal. TREG- Uses no T registers for user variables
U		U-	U+ Limits input to 72 significant columns (UPDATE) U- Limits input to 120 significant columns

Table 2-1. Compiler Options (continued)

Option	Res	Default	Action	
			U*n*	Limits input source width to *n* significant columns; values for *n* can range from 1 to 140.
V	*	V+	V+	Attempts to vectorize FOR loops automatically
			V-	Does not vectorize FOR loops
VI		n/a	VI	Ignores potential vector dependencies while vectorizing the next FOR loop. VI cannot appear on the control statement.
VN		n/a	VN	Does not vectorize the next FOR loop. VN cannot appear on the control statement.
X	*	X-	X+	Collects global cross-reference information
			X-	Does not collect cross-reference information. X+ is the same as XV+:XP+ and X- is the same as XV-:XP-.
XI	*	XI-	XI+	Includes identifier information in the listing
			XI-	Does not include identifier information in the listing
XP	*	XP-	XP+	Generates a global procedure cross-reference
			XP-	Does not generate a global procedure cross-reference
XV	*	XV-	XV+	Generates a global name cross-reference
			XV-	Does not generate a global name cross-reference
Z	*	Z+	Z+	Generates reentrant code
			Z-	Generates code that is not necessarily reentrant

Examples:

```
PASCAL,I=PASCJOB,L=LISTPAS,O=A+:D+:X+:P+:U+.
```

This statement tells the compiler that the source code is in the file PASCJOB. The listing output goes to the file LISTPAS, while the binary output goes to the default file $BLD. The compiler options specified by the O parameter prevent the program from executing if a compilation error is discovered, provide debug information for stack walkbacks, enable the cross-reference listing, put a page break at the beginning of every procedure and function, and restrict the input lines to 72 columns. Default values are accepted for the remaining compiler options.

```
PASCAL,O=ISO:H+24:S+12:V+:Z+.
```

This statement accepts the default file names for the input dataset, the listing dataset, and the binary dataset. The ISO argument to the O parameter disables all extensions provided by CRI. The other compiler options increase the heap memory space by 24000_8 words, increase the stack size by 12000_8 words, enable automatic vectorization of FOR loops, and enable the generation of reentrant code.

2.4 LISTABLE OUTPUT

Pascal produces a dataset that optionally includes the following:

- Page header lines
- Source statement listing
- Error messages
- Program cross-reference
- Procedure and function listing
- Information about identifiers used within a procedure
- Pseudo-CAL listing of the program or module

The Pascal invocation statement and compiler directives allow you to control output and specify the receiving dataset. The actual information included in the output dataset is dependent on the combination of compiler options that are enabled or disabled on the Pascal invocation statement. See table 2-1 for a list of the available options.

Listable output is divided into pages. Under COS, the number of lines per page is controlled by the LPP parameter on the OPTION control statement. See the COS Version 1 Reference Manual for more information about the LPP parameter.

2.4.1 PAGE HEADER LINES

Each page of listable output contains a header line that includes the following information:

- The compiler level
- The time and date compilation began
- The name of the program or module
- The name of the procedure or function being processed for the source statement listing, or the listing title for special listing options
- The global page number

An optional subtitle of up to 72 characters, requested through the ST compiler option, appears on the line following the page header. The current values of the compilation options are also written on this line.

2.4.2 SOURCE STATEMENT LISTINGS

The source statement listing is generated when the L+ list option is selected. The listing is a record of all Pascal statements comprising the program or module as they are sequentially read and interpreted from the source input dataset. A line number is listed to the left of each line identifying its position in the program or module.

The nesting level of a statement appears to the left of the line number. This number reflects the nesting level at the beginning of the line. The program body and each procedure or function body begins and ends with a nesting level of zero.

2.4.3 ERROR MESSAGES

Errors encountered during the compilation of a statement are flagged by lines subsequent to that statement, with error codes listed at the end of the source statement listing. When no listing dataset has been requested, statements with errors and the corresponding error messages are written to the standard output dataset.

2.4.4 CROSS-REFERENCE INFORMATION

Compiler options X+, XV+, and XP+ provide cross-reference information in the Pascal listing as follows:

- XV+ generates a global identifier cross-reference.
- XP+ generates a global procedure cross-reference.
- X+ generates a global identifier cross-reference and a global procedure cross-reference and is the same as specifying XP+:XV+.

2.4.4.1 Global identifier cross-reference

The global identifier cross-reference lists the line numbers in which each identifier appeared in the program or module along with a code giving the type of use. The following codes are used in these references.

Code	Significance
D	Defined or declared
L	Location of label in code
R	Other reference
gR	Referenced label in GOTO statement
wR	WITH statement variable, pointer, or field
S	Set to a new value
pS	Pointer whose referenced variable is changed or passed as a VAR parameter
vS	Passed as a VAR parameter

Only the first 32 characters of an identifier are listed. The first 16 characters of the names of the procedures containing the line numbers also appear in the cross-reference table.

2.4.4.2 Procedure cross-reference

The procedure cross-reference lists information about the program or module and each procedure or function in alphabetical order by name. The procedure cross-reference includes the following information:

- The line numbers of the declaration and body
- The relative address of the entry point

- The number of stack words for this procedure
- The static nesting level; the number of procedures in which this procedure is enclosed (a program or module is at level zero)
- The name of the procedure in which this procedure was declared
- A list of procedures declared in this procedure
- A list of procedures called by this procedure
- A list of procedures that call this procedure

2.4.5 PROCEDURE AND FUNCTION LIST

A list of procedures and functions is provided whenever a source listing, a procedure cross-reference, a pseudo-CAL listing, or identifier information is requested. Procedures and functions are listed in declaration order with the line number of the declaration. Each procedure name is nested according to its static nesting level.

2.4.6 IDENTIFIER INFORMATION

Information about global and local identifiers is written to the listing when the XI+ compiler option is specified. This information is broken apart and stored in the following tables for each procedure or function:

- Types and fields
- Constants
- Local variables
- Nonlocal identifiers

Identifier information for each procedure appears with the procedure cross-reference.

2.4.6.1 Types and fields

The type table contains the following information about each type defined in the procedure:

- The first 32 characters of the name
- The size in words or bits of a variable of this type

- A list of fields within a record type that includes the following:

  - The first 32 characters of the field name
  - The range of words and bits from the record for this field
  - The first 16 characters of the name of the field's type

- A list of constants in an enumerated type with the ordinal value of each constant

2.4.6.2 Constants

The constant table contains the following information about each constant defined in the procedure:

- The first 32 characters of the name
- The size of the constant in words
- The value of the constant

2.4.6.3 Local variables

The local variable table contains the following information about each variable declared in the procedure:

- The first 32 characters of the name

- The block or storage class; one of the following:

  - EXPORTED for exported variables
  - PARAM for formal parameters
  - STACK for local stack variables
  - STATIC for static variables
  - An external name for variables that are imported, in common blocks, or in CACHE†

  Static and exported variables are in a load block with the external name of the program or module.

- Address (octal word offset within the stack frame, octal word offset within the load block, or the formal parameter number); no address is given for imported or common variables.

- The register to which a variable is assigned (the register field can be blank)

- The size in words of the variable

- The first 16 characters of the name of the variable's type

---

† Available with CRAY-2 Computer Systems only

2.4.6.4 Nonlocal identifiers

The nonlocal identifier table includes an alphabetical listing of all nonlocal identifiers, constants, and types referenced in the procedure along with the names of the procedures in which they were declared or defined.

2.4.7 PSEUDO-CAL LISTING

A pseudo-CAL listing of a Pascal program or module is written to the listing dataset when the C+ option is specified on the invocation statement. The CAL instructions displayed by the C+ option are equivalent to the code generated for the program or module after all optimization and scheduling have been performed.

Each statement in the pseudo-CAL listing is organized as follows:

- Relative address of the CAL instruction
- Octal equivalent of the CAL instruction
- The pseudo-CAL instruction generated
- The first 8 characters of the name of the procedure or function in which the instruction appears
- The line number of the Pascal source statement that generated the instruction

Symbolic names are used for compiler-generated labels and for externals. Relative addresses are used for static and exported variables and for literals. Symbolic names are shown added to @MAIN, a local pseudonym for a load block having the external name of the program or module.

2.5 CPU TARGETING†

The CPU parameter allows you to specify characteristics of the machine on which the compiled program executes.

The Pascal compiler, by default, assumes that the compiled program executes on the machine on which it is compiled. The CPU parameter specifies a target machine that differs from the host machine or disables certain characteristics of the host machine.

---

† Not available on the CRAY-2 Computer System

For a description of the target machines and their optional characteristics, see the CPU parameter on the Pascal control statement in this section.

3. PASCAL VOCABULARY

The Pascal vocabulary consists of the following elements:

- Special symbols
- Predefined identifiers
- User-defined identifiers
- Numbers
- Character strings
- Statement labels
- Comments

3.1 SPECIAL SYMBOLS

Special symbols in Pascal fall into three categories:

- Word symbols (also called *reserved words*)
- Operators
- Delimiters

3.1.1 RESERVED WORDS

Pascal reserves certain words for specific uses. These words cannot be redefined by the user or used in any manner other than the ones for which they are intended. The reserved words are as follows:

AND	DOWNTO	GOTO	OR	THEN
ARRAY	ELSE	IF	OTHERWISE	TO
BEGIN	END	IMPORTED	PACKED	TYPE
BY	EXPORTED	IN	PROCEDURE	UNTIL
CACHE†	EXTERNAL	LABEL	PROGRAM	VALUE
CASE	FILE	MOD	RECORD	VAR
COMMON	FOR	MODULE	REPEAT	VIEWING
CONST	FORTRAN	NIL	SET	WHILE
DIV	FORWARD	NOT	STATIC	WITH
DO	FUNCTION	OF	TASKVAR††	

† Available with CRAY-2 Computer Systems only
†† Available with CRAY-1 and CRAY X-MP Computer Systems only

The meaning of each reserved word is discussed in the context of the statement in which it appears.

All words in the foregoing list are standard Pascal reserved words except BY, CACHE†, COMMON, EXPORTED, EXTERNAL, FORTRAN, FORWARD, IMPORTED, MODULE, OTHERWISE, STATIC, TASKVAR††, VALUE, and VIEWING. (In standard Pascal, FORWARD is a directive and not a reserved word.)

3.1.2 OPERATORS

Table 3-1 lists the Pascal operators and gives a brief description of the function of each. Some operators are used in more than one way.

Table 3-1. Pascal Operators

Operator	Description
+	The + operator performs the following functions: • Adds two numbers • Identifies a number as positive • Describes the union of two sets. (See section 5, Data Types, for a description of sets.)
-	The - operator performs the following functions: • Subtracts one number from another • Inverts the sign of a number • Describes the difference of two sets
*	The * operator performs the following functions: • Multiplies two numbers • Describes the intersection of two sets
/	Divides one number by another without truncating the result
DIV	Divides one integer by another and truncates the remainder
:=	Assigns a value to a variable

† Available with CRAY-2 Computer Systems only
†† Available with CRAY-1 and CRAY X-MP Computer Systems only

Table 3-1. Pascal Operators (continued)

Operator	Description
MOD	Gives the remainder resulting from one integer being divided by another (modulo)
AND	Gives the logical conjunction of two Boolean operands
OR	Gives the logical disjunction of two Boolean operands
NOT	Inverts the value of a Boolean operand
=	Tests whether the first operand is equal to the second
<	Tests whether the first operand is less than the second
>	Tests whether the first operand is greater than the second
<=	The <= combination of operators performs the following functions: • Tests whether the first operand is less than or equal to the second • Tests whether all set members of the first operand are also set members of the second operand • Denotes that the first Boolean operand implies the second
>=	The >= combination of operators perform the following functions: • Tests whether the first operand is greater than or equal to the second • Tests whether all set members of the second operand are also set members of the first operand
<>	The <> combination of operators perform the following functions: • Tests whether the first operand is not equal to the second • Gives the exclusive OR result of Boolean operands

Table 3-1. Pascal Operators (continued)

Operator	Description
IN	Tests whether the first operand is a member in the set of the second operand
^	Indicates that the preceding variable is a pointer to a value rather than a value itself

Operands of Boolean operators are evaluated from left to right, and in Cray Pascal evaluation stops as soon as the expression is resolved as TRUE or FALSE.

Example:

```
IF (i<>NIL) AND (t(j)=7) THEN ...
```

If i has a value of NIL in the above example, evaluation stops before t(j)=7 is evaluated, since the expression could only resolve to FALSE.

3.1.3 DELIMITERS

Pascal recognizes the following delimiters:

Blank . , ; : ' ( ) [ ] { } .. (* *) (. .) EOL

The (. and .) delimiters are alternate representations for [and], respectively. Blanks, end-of-line (EOL) characters, semicolons, and comments are separators. At least one separator must occur between reserved words, identifiers, or values. A separator cannot occur within a reserved word, identifier, or value.

NOTE

Cray Pascal does not support the @ character as an alternate representation of the ^, as specified by the ISO Level 1 Pascal standard.

A statement can cross line boundaries. Also, since Pascal source code is free format, more than one statement can be placed on a single line. The semicolon separates both simple and compound statements from each other. A compound statement is a structure in which the reserved words BEGIN and END delimit at least one simple statement.

The general form of a compound statement is as follows:

```
BEGIN statement { ";" statement } END
```

A semicolon need not appear after the final statement, because that statement is followed by the reserved word END rather than another statement. A semicolon would appear after the END if the compound statement that it terminated was followed by another statement, either simple or compound. If, however, a semicolon were placed after the final statement, the compound statement would still execute as expected. Pascal assumes the presence of a null statement following the semicolon in such a case.

3.2 PREDEFINED IDENTIFIERS

Pascal predefines a set of identifiers to refer to data types, files, constants, functions, and procedures. You can redefine these identifiers. The predefined identifiers can be thought of as having been defined in a hypothetical block that surrounds the program. The predefined identifiers are as follows:

$DEBUG	CHR	INTEGER	POP	SQR
ABS	CONNECT	LN	PRED	SQRT
ALFA	COS	LOC	PRODUCT	SUCC
ALL	COSH	LOG	PUT	SUM
ANY	DISPOSE	LSHIFT	READ	TAN
ARCCOS	EOF	MAXINT	READLN	TANH
ARCSIN	EOLN	MAXVAL	REAL	TEXT
ARCTAN	EXP	MINVAL	RESET	TRUE
BAND	FALSE	NEW	REWRITE	TRUNC
BNOT	GET	ODD	ROUND	UNPACK
BOOLEAN	HALT	ORD	RSHIFT	WRITE
BOR	I24	OUTPUT	SIN	WRITELN
BXOR	I32	PACK	SINH	
CHAR	INPUT	PAGE	SIZEOF	

In the previous list, the following are CRI extensions to the ISO standard: ALFA, ALL, ANY, ARCCOS, ARCSIN, BAND, BNOT, BOR, BXOR, CONNECT, COSH, $DEBUG, HALT, I24, I32, LOC, LOG, LSHIFT, MAXVAL, MINVAL, POP, PRODUCT, RSHIFT, SINH, SIZEOF, SUM, TAN, and TANH.

3.3 USER-DEFINED IDENTIFIERS

A user-defined identifier cannot be the same as a reserved word; if it is the same as a predefined identifier, it redefines that identifier.

Identifiers must begin with a letter and may be followed by any number of letters and numbers. Both uppercase and lowercase letters can be used interchangeably. Under extended Cray Pascal, identifiers can start with a letter or any of the following characters:

```
$  %  @
```

Subsequent characters can also include the underscore character under extended Cray Pascal.

The following special characters for use in identifiers are CRI extensions to the ISO Level 1 Pascal standard: $ % @ _.

All characters in an identifier are significant; that is, the compiler does not know an identifier only by its first *n* characters. An identifier must not be continued, however, over two lines, because an end-of-line (EOL) character will appear within it and separate it into two identifiers. Thus, a Pascal identifier is effectively limited to the maximum line size, which is 140 characters.

The following are valid identifiers:

```
X
$Pasc_task1
CrayComputerProcedureNumber2
return2beginning
```

Since the Pascal compiler does not distinguish between uppercase and lowercase letters, all of the following refer to the same variable:

```
IsItTrue
ISITTRUE
isittrue
IsitTRUE
```

3.4 NUMBERS

Numbers in Pascal can be either decimal or octal. Octal numbers are followed by B or b and are restricted to integers.

Numbers cannot contain commas and cannot be extended over a line boundary, but they can be preceded by a plus or minus sign. A real number must either have at least 1 digit on each side of the decimal point or be written in scientific notation.

In scientific notation, the letter E (or e) followed by a scale factor appears at the end of a number. The scale factor can be preceded by a + or -; if neither is present, + is assumed. The E represents: times 10 to the power of. Thus, 2E6 is 2 times 10 to the power of 6, or 2,000,000.

The predefined identifier MAXINT specifies the largest valid positive integer. The value of MAXINT on a Cray computer is $2^{63}-1$ (9,223,372,036,854,775,807 in decimal). -MAXINT is $-2^{63}-1$.

The following are valid numbers:

```
1          12e10
+02        77B (* = 63 *)
-0.6       -71B (* = -57 *)
0.3E-8
```

Octal numbers are a CRI extension to the ISO Level 1 Pascal standard.

3.5 CHARACTER STRINGS

Strings contain one or more characters enclosed in apostrophes. An apostrophe within a string is represented as two apostrophes.

The following examples are valid character strings:

```
'X'
'@'
'Part 1'
'DON''T'
```

A string must be completely contained on a single line.

3.6 STATEMENT LABELS

A statement label serves as the target for a transfer of program control initiated in a GOTO statement (see section 8, Assignment Statement and Program Control Statements). Each label must be declared in the

declarations section of the program block in which it occurs (see section 4, Program Organization). A label is a sequence of 1 to 4 digits.

The following are valid labels:

```
5
75
0
9999
0001
```

3.7 COMMENTS

Comments contain explanatory information that does not generate executable code. Comments are begun by either { or (* and closed by either } or *).

Example:

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

A CRI extension permits comments to be nested by requiring that a comment be closed with the symbol corresponding to the one with which it was opened. If the ISO compiler option is disabled (see section 2.3), a comment opened with a { must be terminated with a }, and a comment opened with a (* must be terminated with a *).

The ability to nest comments is a CRI extension of the ISO Level 1 Pascal standard.

If the ISO compiler option is enabled, a comment that begins with a { can be closed by a *); likewise, a } can close a comment begun by a (*.

This manual uses the (* and *) form of commenting.

A comment beginning with (*# indicates a compiler directive. All compiler directives occurring in the source program must appear in comments. Subsection 2.3 describes compiler directives.

Example:

```
(*#L-*)
```

4. PROGRAM ORGANIZATION

A Pascal program consists of a program heading and a *block* of statements. The main program block may contain other blocks, each of which is either a function or a procedure called from the main program. Each of these inner blocks, in turn, may contain blocks of its own. Figure 4-1 is an example of the organization of a Pascal program.

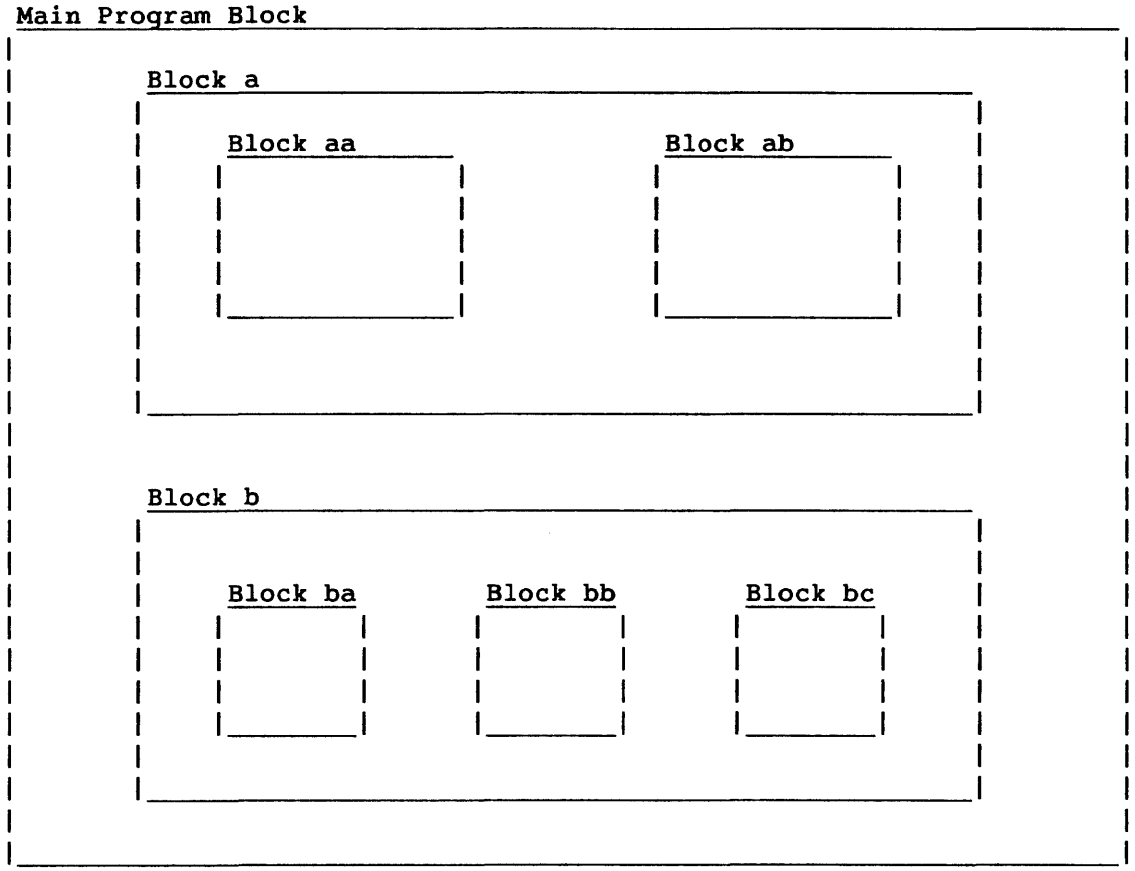

Figure 4-1. Sample Organization of a Pascal Program

The organization of the main program consists of the following parts:

- Heading
- Declarations section
- BEGIN
- Executable statements section
- END.

When the reserved word END signals the end of the main program, it is followed by a period. Comments can appear after the final END and before the initial heading.

4.1 PROGRAM HEADING

The program heading is the first statement in a Pascal program, although it can be preceded by comments. The program heading gives the program a name and lists the optional program parameters (files). The format for the program heading is as follows:

```
program-heading = "program" id [ "(" program-parms ")" ] .

program-parms = id-list .
```

Example:

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

In the example, the program named sample has two parameters. INPUT is a predefined Pascal file. Its use as a parameter in the program heading of a COS job indicates a file of input data as part of the $IN dataset. In UNICOS, INPUT is associated with standard input. INPUT becomes the default when an input file is referenced in the program without being explicitly named.

When OUTPUT is named as a PROGRAM parameter of a COS job, a file is created for the program's output and attached to the $OUT dataset. In UNICOS, OUTPUT is associated with standard output. OUTPUT becomes the default file name when an output file is referenced in the program without being explicitly named.

If identifiers other than INPUT and OUTPUT are listed as program parameters, they must be declared in the declarations section of the program block. Such a file is bound to a COS local dataset or a UNICOS file in the current directory of the same name. A COS dataset becomes local to the program when it is named in an ACCESS, ACQUIRE, or ASSIGN control statement in the job control language (JCL) file. (See section 2, Using Pascal on a Cray Computer, for a description of the JCL file.) The predefined procedure CONNECT (see section 10, Input and Output) permits a name other than the COS local dataset name or the name of a UNICOS file in the current directory to be assigned to the file within the scope of the Pascal program. The order in which the parameters are listed is not relevant.

NOTE

Identifiers for program parameters are limited to 7 characters and cannot include the underscore character, but can start with %, @, or $.

COS example:

```
ACCESS, DN=INFILE ...
ASSIGN, DN=OUTFILE ...
PASCAL.
:
/EOF
PROGRAM figure (infile, outfile);
VAR infile, outfile : FILE OF INTEGER;
```

The VAR declaration used in the example is described later in this section.

4.2 DECLARATIONS SECTION

The declarations section contains declarations and definitions of elements local to the block. These elements cannot be accessed outside of the block in which they are declared, but they can be accessed in any functions or procedures defined in the same block.

The declarations section follows the heading and precedes the reserved word BEGIN, which separates the declarations from the executable statements. The kinds of declarations are as follows:

- Labels
- Constants
- Types
- Variables
- Values
- Procedures and functions

The compiler allows constant, type, variable, and label sections to be intermixed and repeated. For example, constant declarations can appear both before and after type declarations. By reordering declarations, you can place related items together. The following restrictions apply to the reordering of declarations:

- The VALUE section, if present, must follow all constant, type, variable, and label declarations, and precede all procedure and function declarations.

- Procedure and function declarations must come last.

- Forward TYPE references occurring in pointer declarations must be defined before the end of the TYPE section in which they are referenced.

Example:

```
TYPE x = ^b;
     b = RECORD ...
```

The declaration in the preceding example is valid because b is defined in the same TYPE section as x, which references it. The declaration of x would be invalid if a series of variable declarations separated it from the declaration of b.

Intermixing declarations parts is a CRI extension to the ISO Level 1 Pascal standard.

4.2.1 LABEL DECLARATIONS

A label used in the executable statements section of a block as the target of a GOTO statement must first be declared. The format of a label declaration is as follows:

```
label-decl-part = [ "label" label { "," label } ";" ] .
```

A valid label consists of 1 to 4 digits.

Example:

```
LABEL 10, 20;
```

4.2.2 CONSTANT DEFINITIONS

A constant identifier assigned a value in the declarations section of a block retains that value within the scope of the block. The compiler substitutes the value whenever the constant identifier is encountered in the block. The format of the constant definition section is as follows:

```
constant-def-part = [ "const" constant-def ";" { constant-def ";" } ] .
constant-def = id "=" (constant | constant-expression) .
```

A constant identifier can be any valid Pascal identifier (see section 3, Pascal Vocabulary). The constant itself can be a number, a character string, another constant identifier, or a constant expression. If it is a number, the constant can optionally be preceded by a sign.

Example:

```
CONST  pi = 3.1415927;
       maximum = +99999;
       found = FALSE;
```

If the constant is a character string, it takes the following form:

```
character-string = "'" string-element { string-element } "'" .
```

If an apostrophe is one of the elements in the string, it is represented by two apostrophes.

Example:

```
CONST  blank = ' ';
       caution = 'Don''t change mode at this point.';
```

The use of a constant expression in a constant definition allows a constant identifier to be defined as a result of operations on other constants that are already defined. A constant expression has the following format:

```
constant-expression =          constant-subexpression |
                               ("if" constant-subexpression
                               "then" constant-subexpression
                               "else" constant-subexpression ) .

constant-subexression =        simple-constant-expression
                               [ relational-operator
                               simple-constant-expression ] .

simple-constant-expression =   [ sign ] constant-term
                               { adding-operator constant-term } .

constant-term =                constant-factor
                               constant-function-designator |
                               constant-set-constructor |
                               ( "(" constant-expression ")" ) |
                               ( "not" constant-factor ) .

constant-set-constructor =     "[" [constant-member designator
                               { "," constant-member designator } ]
                               "]" .

constant-member-designator =   constant-expression [".."
                               constant-expression ] .
```

Along with allowing you to assign integer, real, Boolean, and string constants, constant expressions let you define constant sets.

All of the predefined functions listed in appendix B, Predefined Functions and Procedures, can be used in constant expressions with the exception of LOC, SIZEOF, and those functions that return the status of files. Function arguments must be constants that have already been defined.

Example:

```
CONST  max1 = 150;
       max2 = 200;
       maxsize = (max1+max2)*5;
       uppercase = ['A'..'Z'];
       lowercase = ['a'..'z'];
       letter = uppercase+lowercase;
       ord0 = ORD('0');
       size = IF max1<100 THEN 'big' ELSE 'small';
       pi = ARCCOS(-1.0);
```

The use of constant expressions is a CRI extension to the ISO Level 1 Pascal standard.

4.2.3 TYPE DEFINITIONS

Data types are defined by the TYPE definition in the declarations section of a Pascal program. (Section 5, Data Types, describes the data types available.) A data type must be declared in the declarations section of a program before it is referenced in an executable statement.

The format of the TYPE definition section is as follows:

```
type-def-part = [ "type" type-def ";" { type-def ";" } ] .

type-def = id "=" type-denoter .

type-denoter = type-id | new-type .

new-type = new-ordinal-type | new-structured-type | new-pointer-type .
```

Any of the following can be created with the TYPE definition:

- A new data type with the same characteristics as another data type. For example:

```
TYPE oranges = INTEGER;
     apples = INTEGER;
```

  Types defined as integers are stored in 64 bits by default. You can create 24- and 32-bit integers by using the I24 and I32 types, which are CRI extensions (see section 5, Data Types). For example:

```
TYPE oranges = I24;
     apples = I32;
```

- An enumerated type. For this type (described more fully in section 5, Data Types), the identifiers in parentheses receive ordinal values based on the order in which they are specified. For example:

```
TYPE days =  (sunday, monday, tuesday, wednesday, thursday,
              friday, saturday);
```

  In the example, sunday has the smallest value (an ordinal number of 0) and saturday the largest (an ordinal number of 6).

- A subrange of another type. For example:

```
TYPE cardslot = 1..80;
     workdays = monday..friday;
     letters = 'a'..'z';
```

- A new structured type. For example:

```
TYPE lineimage = ARRAY [1..80] OF CHAR;
     crayword = PACKED ARRAY [1..8] OF CHAR;
     node = RECORD
              last: INTEGER;
              data: ALFA;
              next: INTEGER
            END;
```

- Pointer type to access dynamically allocated variables. For example:

```
TYPE link = ^node;
     node = RECORD
              data: INTEGER;
              left, right: link
            END;
```

4.2.4 VARIABLE DECLARATIONS

A variable declaration does the following:

- Associates a variable's identifier with a data type
- Reserves a space in memory for the variable
- Can create a new type without explicitly naming it, called an anonymous type

The type denoter, when not a standard type, can be a list of values, a subrange, a structured type, or a pointer type. In addition to the ISO-standard VAR declaration, Cray Pascal supports the EXPORTED, IMPORTED, STATIC, COMMON, TASKVAR,† and CACHE†† variable declarations, which are described in the following subsections.

† Available with CRAY-1 and CRAY X-MP Computer Systems only
†† Available with CRAY-2 Computer Systems only

4.2.4.1 VAR declaration

The VAR declaration associates the identifier for a variable with a data type. Also, you can create new types in the variable declaration without explicitly naming the type (an anonymous type). When the type denoter in the VAR declaration is not a standard type, it can be a list of values, a subrange, a structured type, or a pointer type.

VAR variables declared at the program or module level are allocated in static storage; all other VAR variables are allocated on the run-time stack.

The format of a VAR variable declaration section is as follows:

```
var-dcl-part = [ "var" var-dcl ";" { var-dcl ";" } ] .

var-dcl = id-list ":" type-denoter .
```

An error message is issued at compile time in the following cases:

- A variable is declared as one type and treated as another type in an executable statement.
- A variable is used in an executable statement but is not declared.

Example:

```
VAR count, current, i : INTEGER;
x, result : REAL;
done : BOOLEAN;
table : ARRAY [1..1000] OF REAL;
day : (sunday, monday, tuesday, wednesday, thursday, friday,
      saturday);
letter : 'a'..'z'
```

The last three declarations shown create new types. Variables declared in a list (such as count, current, and i in the previous example) are equivalent types. The value of one variable can always be assigned to any other variable in the list. Structurally compatible variables that appear in different declarations, however, are not equivalent.

Example:

```
VAR ap,bp,cp : ^b;
    dp       : ^b;
BEGIN
    ap := bp;            (* This assignment is valid *)
    ap := dp;            (* This causes a type mismatch error *)
```

4.2.4.2 EXPORTED declaration

EXPORTED and IMPORTED variable declarations allow different compilation units to share variables. EXPORTED defines a variable and allows other compilation units to access it with the IMPORTED declaration. Only one unit can export a given variable, but any number of other units can import it. Storage for a variable defined as IMPORTED is not allocated in the importing module; references to the imported variable are set up as external references and linked to the EXPORTED variable when the loader is run. See section 10, Input and Output, for a description of modules.

Exported variables are allocated in static storage by the compiler with CAL-style entry names. If an alias is supplied, it is used as the external name. If no alias is supplied, the variable name, which must be 8 characters or less, is used as an external name. Declaring an exported variable in a recursive routine causes the routine always to access the same memory words for the variable, regardless of which recursive scope is active. Pascal allows exported variables in VALUE statements.

An error message is issued at compile time in the following cases:

- A variable is declared as one type and treated as another type in an executable statement.
- A variable is used in an executable statement but is not declared.
- The external name of the variable contains more than the maximum number of characters permitted: 32 characters for CRAY-2 Computer Systems and 8 characters for CRAY X-MP and CRAY-1 Computer Systems.
- Two variables are exported with the same external names.

The format of an EXPORTED variable declaration section is as follows:

```
exported-var-dcl-part = [ "EXPORTED" ext-var-dcl ";"
                        { ext-var-dcl ";" } ] .

ext-var-dcl =  id [ "(" external-name ")" ]
               { "," id [ "(" external-name ")" ] } : type-denoter.
```

Example:

```
EXPORTED count (CNT), current(CURRENT), i(EYE) : INTEGER;
    done : BOOLEAN;
    table : ARRAY [1..1000] OF REAL;
    day : (sunday, monday, tuesday, wednesday, thursday, friday,
    saturday);
    letter : 'a'..'z'
```

In the previous example, the last three declarations create new types. The variables count, current, and i will appear in the entry list under the respective names CNT, CURRENT, and EYE. Done, table, day, and letter will appear in the entry list as DONE, TABLE, DAY, and LETTER.

The EXPORTED variable is a CRI extension to the ISO Level 1 Pascal standard.

4.2.4.3 IMPORTED declaration

IMPORTED allows a compile unit to access a variable that was previously declared in another compilation unit as EXPORTED. The loader issues an error message if an imported variable was never exported. Pascal does not allow imported variables in VALUE statements.

Imported variables are not allocated by the compiler; instead a CAL-style external name is associated with the variable. This name is used as the label for the space allocated to the variable. The routine exporting the variable allocates this space and labels it with the external name so that the loader can resolve references to the name. If no alias is supplied, the external name is the variable name; if an alias is supplied, it is used as the external name.

Declaring an imported variable in a recursive routine causes the routine's code to access the same memory words for the variable, regardless of which recursive scope is active. Storage can be shared by routines with disjoint scope by exporting a variable from one procedure and importing where needed.

An error message is issued at compile time in the following cases:

- A variable is declared as one type and treated as another type in an executable statement.
- A variable is used in an executable statement but is not declared.
- The external name of the variable contains more than the maximum number of characters permitted: 32 characters for CRAY-2 Computer Systems and 8 characters for CRAY X-MP and CRAY-1 Computer Systems.

The format of an IMPORTED variable declaration section is as follows:

```
imported-var-dcl-part = "IMPORTED" ext-var-dcl ";"
                        { ext-var-dcl ";" } .

ext-var-dcl = id [ "(" external-name ")" ]
              { "," id [ "(" external-name ")" ] } : type-denoter.
```

Example:

```
IMPORTED count (CNT), current(CURRENT), i(EYE) : INTEGER;
    done : BOOLEAN;
    table : ARRAY [1..1000] OF REAL;
    day : (sunday, monday, tuesday, wednesday, thursday, friday,
          saturday);
    letter : 'a'..'z'
```

In the previous example, the variables count and current will appear in the external list under the respective names CNT and CURRENT. Done, table, day, and letter will appear in the external list as DONE, TABLE, DAY, and LETTER.

Declaring x(EYE) and i(EYE) causes the same memory word (EYE in the external list) to be used for both variables. Select unique external names to prevent this FORTRAN-style equivalence. When equivalence is desired, it is best done by declaring the variable as a variant record or by using the VIEWING statement.

The IMPORTED variable is a CRI extension to the ISO Level 1 Pascal standard.

4.2.4.4 STATIC declaration

A static variable keeps its value between calls to the procedure that contains it; in this way it resembles a global variable, but it is known only to the procedure in which it is declared. The compiler allocates static variables in static storage. Declaring a static variable in a recursive routine causes the routine's code always to access the same memory words for the variable, regardless of which recursive scope is active. Pascal allows static variables in VALUE statements, but only at the program or module level.

An error message is issued at compile time in the following cases:

- A variable is declared as one type and treated as another type in an executable statement.
- A variable is used in an executable statement but is not declared.

The format of a STATIC variable declaration section is as follows:

```
static-var-dcl-part = [ "STATIC" var-dcl ";" { var-dcl ";" } ] .

var-dcl = id-list ":" type-denoter .
```

Example:

```
STATIC count, current, i : INTEGER;
    x, result : REAL;
    done : BOOLEAN;
    table : ARRAY [1..1000] OF REAL;
    day : (sunday, monday, tuesday, wednesday, thursday, friday,
          saturday);
    letter : 'a'..'z'
```

The STATIC declaration is a CRI extension to the ISO Level 1 Pascal standard.

4.2.4.5 COMMON declaration

The COMMON declaration allows the use of FORTRAN common blocks as a means of sharing data across compile units. Each common variable is allocated by the compiler as a named common block; Pascal does not support blank COMMON. Common variables cannot appear in the VALUE section.

If no alias is supplied, the external common block name is a variable name containing 8 or less characters. If an alias is supplied, it is used as the common block name. Declaring a common variable in a recursive routine causes the routine's code always to access the same memory words for the variable, regardless of which recursive scope is active.

The format of a COMMON variable declaration section is as follows:

```
common-var-dcl-part = [ "COMMON" ext-var-dcl ";" { ext-var-dcl ";" } ] .

ext-var-dcl  = id [ "(" external-name ")" ]
               { "," id [ "(" external-name ")" ] } : type-denoter.
```

An error message is issued at compile time in the following cases:

- A variable is declared as one type and treated as another type in an executable statement.
- A variable is used in an executable statement but is not declared.
- The external name of the variable contains more than the maximum number of characters permitted: 32 characters for CRAY-2 Computer Systems and 8 characters for CRAY X-MP and CRAY-1 Computer Systems.
- More than 125 common blocks are declared in a single program or module.

Example:

```
COMMON count (CNT), current(CURRENT), i(EYE) : INTEGER;
    done : BOOLEAN;
    table : ARRAY [1..1000] OF REAL;
    day : (sunday, monday, tuesday, wednesday, thursday, friday,
           saturday);
    letter : 'a'..'z'
```

The COMMON declaration is a CRI extension to the ISO Level 1 Pascal standard.

4.2.4.6 TASKVAR declaration†

The TASKVAR declaration allows the use of FORTRAN TASK COMMON blocks as a means of sharing data across compile units. TASKVAR variables cannot appear in the VALUE section.

The format of a TASKVAR variable declaration section is as follows:

```
taskvar-var-decl-part = ["taskvar" ext-var-decl ";"
                         {ext-var-decl ";" } ] .
```

If a variable name is not supplied, the external TASK COMMON block name is the 8-character variable name. If a variable name is supplied, it is used as the TASK COMMON block name. Declaring a TASKVAR variable in a recursive routine causes the routine's code always to access the same TASK COMMON memory words for the variable, regardless of which recursive scope is active.

An error message is issued at compile time in the following cases:

- A variable is declared as one type and treated as another type in an executable statement.
- A variable is used in an executable statement but is not declared.
- The external name of the variable contains more than 8 characters.
- More than 125 common blocks are declared in a single program or module. If different block types are included in a single program or module, the combined total of all blocks specified within the program or module cannot exceed 125.

---

† Not available with CRAY-2 Computer Systems

Example:

```
TASKVAR
    localarray(LA): ARRAY [1..100] OF INTEGER;
    taskno : INTEGER;
    table : 'a'..'z';
```

The TASKVAR declaration is a Cray Research extension to the ISO Level 1 Pascal standard.

4.2.4.7 CACHE declaration†

The CACHE declaration allows you to make use of Local Memory, which is available on CRAY-2 Computer Systems only. CACHE variables are allocated by the compiler in FORTRAN-style common blocks. If no alias is supplied, the externalized common block name is the variable name. If an alias is supplied, it is used as the local common block name.

If a CACHE variable is declared in a recursive routine, the routine always accesses the same memory words for the variable regardless of the recursive scope that is currently active.

NOTE

CACHE variables cannot appear in VALUE statements, and they cannot be passed as VAR parameters.

An error message is issued at compile time in the following cases:

- A variable is declared as one type and treated as another type in an executable statement.
- A variable is used in an executable statement but is not declared.
- The external name of the variable contains more than 32 characters.
- A CACHE variable is passed as a VAR parameter to any procedure or function.

† Available with CRAY-2 Computer Systems only

The format of a CACHE variable declaration section is as follows:

```
var-decl-part = [ "cache" ext-var-decl ";" { ext-var-decl } ] .

ext-var-decl = id [ "(" external-name ")" ]
               { "," id [ "(" external-name ")" ] } : type-denoter .
```

Example:

```
CACHE count(CNT), current(CURRENT), i(EYE) : INTEGER;
    x(EYE), result : REAL;
    done : BOOLEAN;
    table : ARRAY [1..1000] OF REAL;
    day : ( sunday, monday, tuesday, wednesday, thursday, friday,
            saturday );
    letter : 'a'..'z';
```

In the previous example, the last two declarations create new types. By declaring x(EYE) and i(EYE), the same memory word is used for both variables. You should use unique common block names to prevent this FORTRAN-style equivalence. If variables are intended to be equivalent, they should be specified with the VIEWING statement or as variant records.

Variables declared in a list (for example, count, current, and i) are equivalent types. The value of one variable can always be assigned to any other variable in a list. Structurally compatible variables that appear in different declarations are not, however, equivalent.

Example:

```
CACHE ap, bp, cp : ^b;
      dp : ^b;
      ep(VERYLONGEXTERNALNAME) : ^b;
    BEGIN
      ap := bp;  (* This assignment is valid *)
      ap := dp;  (* This assignment causes a type mismatch error *)
      .
      .
      .
```

The CACHE declaration is a CRI extension to the ISO Level 1 Pascal standard.

4.2.5 VALUE DEFINITIONS

The VALUE definition initializes data at compile time, so that explicit initial assignments are not needed at run time. VALUE initializes variables to the values supplied, but they are still variables; their values can be changed by subsequent assignments.

The VALUE definition imposes the following restrictions:

- Only program-level or module-level variables can be initialized. Variables declared in a procedure or function cannot be initialized with VALUE.
- Partial initialization is not permitted. That is, an entire structured variable must be initialized if any part of it is initialized.

The VALUE definition must appear immediately after all program-level or module-level declarations of constants, types, and variables. Because VALUE is a Cray extension to Pascal, it is not accepted in ISO mode.

The format for VALUE is as follows:

```
value-def-part =     "value" value-def ";" { value-def ";" } .

value-def =          entire-var "=" value-spec .

entire-var =         var-id .

value-spec =         simple-value-spec |
                     set-value-spec |
                     array-value-spec |
                     record-value-spec .

simple-value-spec =  unsigned-constant |
                     sign unsigned-number |
                     sign constant-id.

set-id =             id .

set-value-spec       ( set-id | "[" [ set-value-elt-list ] "]" ) .

array-value-spec =   [ type-id ]
                     "(" sub-value-spec { "," sub-value-spec } ")" .

record-value-spec =  [ type-id ]
                     "(" value-spec { "," value-spec } ")" .
```

```
set-value-elt=      constant [ ".." constant ] .

set-value-elt-list= set-value-elt { "," set-value-elt } .

sub-value-spec =    value-spec |
                    unsigned-integer "of" value-spec .
```

Simple value specifiers initialize variables of scalar, real, and string types. Each simple value specifier uses the same format as a constant of the same type.

Example:

```
PROGRAM ex0 (OUTPUT);

   VAR
      x : INTEGER;
      y : REAL;
      z : (red, white, blue);
      q : PACKED ARRAY [1..10] OF CHAR;

   VALUE
      x = 4;
      y = 129.375;
      z = white;
      q = 'test1test2';

   BEGIN
   END.
```

Set value specifiers initialize variables of set types. The specifiers look like constant sets.

Example:

```
   PROGRAM ex1 (OUTPUT);

   VAR
      x : SET OF 0..63;
      y : SET OF (red, orange, yellow, green, blue);

   VALUE
      x =  [1, 4, 5..8];
      y =  [red, orange];

   BEGIN
   END.
```

An array value specifier consists of an optional type name followed by parentheses enclosing a comma-separated list of value specifiers. The type name is array type, not the type of the elements of the array. One value specifier is provided for each element of the array, in order from lowest index to highest index. Because what serves as a two-dimensional array in Pascal is actually an array whose elements are arrays, multidimensional arrays should be value-initialized accordingly. Cray Pascal supports the following notation as shorthand for a list of identical value specifiers within an array value specifier:

```
unsigned-integer "of" value_spec
```

Example:

```
PROGRAM ex2 (output);

   TYPE
      t1 = ARRAY [1..4] OF INTEGER;
      t2 = ARRAY [1..2] OF t1;

   VAR
      x : ARRAY [1..4] OF CHAR;
      xn : packed ARRAY [1..4] OF CHAR;
      y : t2;
      q : ARRAY [1..555] OF INTEGER;

   VALUE
      y = t2(t1(1, 2, 3, 4), t1(5, 6, 7, 8));
      xn = ('a', 'b', 'c', 'd');

      (* Note that the above declaration could have been written
         xn = 'abcd', since xn is a string, in agreement with the ISO
         standard for denoting strings.  Also, type names can
         optionally be included in the VALUE statement. *)

   x = ('a', 'b', 'c', 'd');

      (* Note that the above declaration could not have been written
           x = 'abcd';
         since x is not a string, in agreement with the ISO standard
         for denoting strings. *)

   q = (427 of 0, 1, 72 of 4, 55 of 6);

   BEGIN
   END.
```

A record value specifier consists of an optional type name followed by parentheses enclosing a comma-separated list of value specifiers. One value specifier is provided for each field of the record, in declaration order.

In the case of variant records, only one variant may be initialized. The appropriate variant is selected by the compiler based on the values to which the variant-selectors are initialized. Thus, appropriate values must be provided for all variant-selectors on the desired path, even if there is no corresponding tag field.

Example:

```
PROGRAM ex3 (OUTPUT);

VAR
   qwert1, qwert2 : RECORD
                    x, y : INTEGER;
                    q : RECORD
                        r1, r2 : BOOLEAN
                        END;
                    bleem : PACKED ARRAY [1..4] OF INTEGER;
                        CASE BOOLEAN OF
                        TRUE : (harvey : CHAR);
                        FALSE : (max : REAL);
                    END;
VALUE
   qwert1 = (1, 2, (TRUE, FALSE), (5, 4, 1, 3), TRUE, 'x');
   qwert2 = (5, 9, (TRUE, TRUE), (1, 9, 8, 7), FALSE, 1.43);

BEGIN
END.
```

Example:

```
PROGRAM ex4 (OUTPUT);

      CONST
      one = 1;
      fone = 1.0;
      shortstr = 'SHORT';
      longstr = 'LONG(YES, LONG)';
      seven = 7;
      large = 655361435;
```

Example: (continued)

```
TYPE
   settype = SET OF 0..5;

VAR
   int, out, bigtest : INTEGER;
   bleem : 1..6;
   srt, swrng : REAL;
   set1, set2, set3, set4 : settype;
   x, y : PACKED ARRAY [1..5] OF CHAR;

VALUE
   int = -1;
   out = seven;
   bigtest = large;
   bleem = one;
   srt = fone;
   swrng = 2.24536457;
   set1 = [0,1..3,5];
   set2 = [one];
   set3 = [1];
   set4 = [5..4];
   x = 'SILLY';

BEGIN
END.
```

The VALUE definition is a CRI extension to the ISO Level 1 Pascal standard.

4.2.6 PROCEDURE AND FUNCTION DECLARATIONS

Procedures and functions invoked in the executable statements section of a given block must be defined in the declarations section of that block.

The format for the procedure and function declarations section is as follows:

```
procedure-and-function-declaration-part =
  { ( procedure-declaration } function-declaration ) ";" } .
```

The organization of procedures and functions parallels that of the main program. Section 9, Procedures and Functions, describes procedures and functions more fully.

5. DATA TYPES

Pascal data types fall into the following three categories:

- Scalar, which includes the following:
  - Integer
  - I24
  - I32
  - Real
  - Boolean
  - Character
  - Enumerated
  - Subrange
- Structured, which includes the following:
  - Array
  - ALFA
  - Record
  - Set
  - File
- Pointer

I24, I32, and ALFA are CRI extensions to the ISO Level 1 Pascal standard.

A scalar type is composed of an ordered group of constants. Because they are ordered in sequence, constants of the same scalar type or of equivalent types can be tested as to their relationships to each other. One constant is always either equal to, greater than, or less than another constant of the same scalar type.

A structured type is composed of scalar types or of other structured types. Users can manipulate a structured type to store complex data in a logical and convenient manner.

A pointer type accesses elements of a different type, including dynamically allocated variables.

5.1 REPRESENTATION OF SCALAR TYPES

Variables of the standard Pascal scalar types are stored in one 64-bit word, except when the qualification PACKED is included in the declaration. (Packed structures are described later in this section.)

However, the data field width of integers, characters, and Boolean values is by default 64 bits, 8 bits, and 1 bit, respectively. The data is right-justified in the word.

The number of bits used to store each scalar data type in a packed structure is as follows:

Data Type	Number of Bits
Boolean	1
Character	8
Subrange (*a..b*)	*n*
Enumerated	*n*
Integer	64
I24	24
I32	32
Real	64

In the previous list, *n* is the number of bits required to hold the largest *ordinal number* in the data-type. An ordinal number is the internal integer representation of that value. All scalar data types except real have ordinal numbers.

5.2 INTEGER TYPE

The integer type includes all whole numbers within the closed range of -MAXINT through MAXINT. (Section 3, Pascal Vocabulary, describes MAXINT and numbers that qualify as valid integers.) The ordinal number of an integer is the integer value itself.

Any integer variables used in a Pascal program must be declared in the declarations section of the relevant block (see section 4, Program Organization).

Table 5-1 describes the operations for integers. The variables x and y are assumed to have been declared as integers and assigned values of 4 and 2, respectively. These operations are also valid if one or both operands are of type I24 or I32 (described later).

Table 5-1. Integer Operations

Operation	Result	Description
x + y	6	Adds y to x
x - y	2	Subtracts y from x
x * y	8	Multiplies x times y
x / y	2.0	Divides y into x, giving a real number for the result. An error occurs if y is 0. (Pascal converts both operands to real numbers in-line before executing this operation.)
x DIV y	2	Divides y into x, giving a truncated integer for the result. An error occurs if y is 0.
x MOD y	0	Returns the remainder after dividing y into x. An error occurs if y is equal to or less than 0.
x = y	FALSE	Tests for x equal to y
x > y	TRUE	Tests for x greater than y
x < y	FALSE	Tests for x less than y
x >= y	TRUE	Tests for x greater than or equal to y
x <= y	FALSE	Tests for x less than or equal to y
x <> y	TRUE	Tests for x not equal to y

The following predefined Pascal functions return integer results. In the following list, *i* and *j* represent variables or expressions of type INTEGER, *r* represents a variable or expression of type REAL, and *e* represents an element in any scalar type except type REAL.

Function	Description
ABS(*i*)	Returns the absolute value of *i*
BAND(*i*,*j*)	Determines the logical product (AND) of *i* and *j*
BNOT(*i*)	Determines the logical ones complement of *i*

Function	Description
BOR(*i,j*)	Determines the logical inclusive OR of *i* and *j*
BXOR(*i,j*)	Determines the logical exclusive OR of *i* and *j*
LSHIFT(*i,j*)	Shifts the word *i* left *j* bit positions
POP(*i*)	Returns the population count (number of bits set to 1) in the word *i*
RSHIFT(*i,j*)	Shifts the word *i* right *j* bit positions.
ORD(*e*)	Returns the ordinal number of the element *e*
PRED(*i*)	Returns the integer preceding the integer *i* (*i* - 1)
ROUND(*r*)	Rounds *r* to the nearest integer
SQR(*i*)	Squares *i* (*i* * *i*)
SUCC(*i*)	Returns the integer succeeding the integer *i* (*i* + 1)
TRUNC(*r*)	Truncates the fractional part of *r* and returns an integer

---

BAND, BNOT, BOR, BXOR, LSHIFT, POP, and RSHIFT are CRI extensions to the ISO Level 1 Pascal standard.

5.3 I24 AND I32 TYPES

The I24 and I32 data types store integers in 24 and 32 bits, respectively, rather than the 64 bits used by type INTEGER. Multiplication and division operations are significantly faster with operands of type I24 and type I32 than with operands of type INTEGER. While both I24 and I32 are available on all machine lines, I24 offers the best speedup on the CRAY X-MP and the CRAY-1 Computer Systems and I32 on CRAY-2 Computer System.

Types I24 and I32 are CRI extensions to the ISO Level 1 Pascal standard.

The types I24 and I32 are defined as a subrange, as in the following declarations:

```
TYPE I24 = -8388607..8388607;
TYPE I32 = -2147483647..2147483647;
```

If a value outside of these ranges is assigned to an I24 or I32 variable and if run-time error checking is enabled, the program aborts with an error message. No constant corresponding to MAXINT exists for the maximum value for variables of type I24 or I32.

The same operations available to operands of type INTEGER can be used when one or both operands are of type I24 or I32. Table 5-1 describes these operations.

5.4 REAL TYPE

Data elements of type REAL are represented either in scientific notation (as in 8E5) or with a decimal point. (See section 3, Pascal Vocabulary, for examples of both types.) A number written in scientific notation is considered a real number whether or not a decimal point is present.

A variable must be declared as type REAL in the declarations section of a block before it can be used in the executable statements section.

The range of valid numbers of type REAL is from 2^{-8191} to $2^{8191}-1$. Real numbers are precise up to 48 binary digits (approximately 14 decimal digits).

Table 5-2 describes the arithmetic operations for real numbers. The variables x and y are assumed to have been declared as type REAL and assigned values of 4.8 and 2.4, respectively.

Table 5-2. Real Number Operations

Operation	Result	Description
x + y	7.2	Adds y to x
x - y	2.4	Subtracts y from x
x * y	11.52	Multiplies x times y
x / y	2.0	Divides y into x. A fatal error occurs if y is 0.0.

Relational operators (>, <, <>, <=, and >=) can be applied to operands of type REAL just as to operands of type INTEGER (see table 5-1). For example, assuming the same values for x and y as in table 5-2, the following operations both yield TRUE as a result:

```
x >= y
x <> y
```

NOTE

Comparing real variables for exact equality (x=y) may not always yield the expected result, since most values can only be represented approximately. Errors in representation and computation of real numbers may cause a comparison to fail for variables that are approximately equal.

The following predefined functions return values of type REAL. *r* represents a real variable or expression.

Function	Description
ABS(*r*)	Returns the absolute value of *r*
ARCCOS(*r*)	Calculates the inverse of the cosine of *r*
ARCSIN(*r*)	Calculates the inverse of the sine of *r*
ARCTAN(*r*)	Calculates the arctangent of *r*
COS(*r*)	Calculates the cosine of *r*
COSH(*r*)	Calculates the hyperbolic cosine of *r*
EXP(*r*)	Calculates the exponential function of *r*
LN(*r*)	Determines the natural logarithm of *r*
LOG(*r*)	Calculates the common logarithm of *r*
SIN(*r*)	Calculates the sine of *r*
SINH(*r*)	Calculates the hyperbolic sine of *r*
SQR(*r*)	Squares *r*
SQRT(*r*)	Calculates the square root of *r*
TAN(*r*)	Calculates the tangent of *r*
TANH(*r*)	Calculates the hyperbolic tangent of *r*

LOG, TAN, ARCSIN, ARCCOS, SINH, COSH, and TANH are CRI extensions to the ISO Level 1 Pascal standard.

5.5 BOOLEAN TYPE

The Boolean data type has two constants: TRUE and FALSE. They are ordered such that TRUE, with an ordinal number of 1, is greater than FALSE, with an ordinal number of 0. Variables that take on Boolean values in the executable statements section of a block must be declared as type BOOLEAN in the declarations section of that block.

All operations involving relational operators (<, >, <>, <=, and >=.) yield Boolean results. The operands can be scalar types in all cases and pointer or set types in some cases. See the descriptions of types POINTER and SET in this section and the description of operators in section 3, Pascal Vocabulary, for information on valid operations.

Table 5-3 lists the operations that take Boolean operands and yield Boolean results. Variables x and y are assumed to have been declared as Boolean variables and assigned values of TRUE and FALSE, respectively.

Table 5-3. Boolean Operations

Operation	Result	Description
x AND y	FALSE	Finds the logical conjunction of x and y. Both operands must be TRUE to yield a result of TRUE.
x OR y	TRUE	Finds the logical disjunction of x and y. If either operand is TRUE, the result is TRUE.
NOT x	FALSE	Returns the value that is the opposite of the value of x

The following predefined functions return values of type BOOLEAN. *i* represents an integer and *fn* represents a file name.

Function	Description
EOF(*fn*)	Returns a value of TRUE if an end-of-file condition exists for *fn*
EOLN(*fn*)	Returns a value of TRUE if an end-of-line condition exists for *fn*

Function	Description
ODD(*i*)	Returns a value of TRUE if the integer is an odd number
PRED(TRUE)	Returns a value of FALSE
SUCC(FALSE)	Returns a value of TRUE

5.6 CHAR TYPE

The constants in the character (CHAR) data type are the characters, both printable and nonprintable, in the ASCII character set (see appendix A, Character Set).

A character is designated as a member of type CHAR by surrounding it with apostrophes. When the character to be designated is an apostrophe, it is represented by two apostrophes.

Examples:

```
'x'
'8'
''''
```

Constants of type CHAR are always single characters. Constructs containing multiple characters surrounded by apostrophes are called *strings*. String data types are described later in this section.

The character constants are mapped to a consecutive series of ordinal numbers. The ordering relationship between any two characters is the same as between their ordinal numbers. For instance, the ordinal numbers for 'A' and 'B' are 65 and 66, respectively. The ordinal numbers for constants of type CHAR are specified in decimal and appear in appendix A, Character Set, under the ASCII decimal code column.

Operands of type CHAR can be used with relational operators in the same manner as integers (see table 5-1). In the following example, the variable ch is assumed to have been declared as type CHAR. If both operations yield a result of TRUE, the value of ch is an uppercase letter. (The ASCII character set is assumed in this example. If these tests were made using an EBCDIC character set, the value of ch would not necessarily be an uppercase letter.)

Examples:

```
ch >= 'A'
ch <= 'Z'
```

The following predefined functions return values or accept parameters of type CHAR. *c* represents a character variable or expression and *i* represents an integer variable or expression. (The ASCII character set is assumed.)

Function	Description
CHR(*i*)	Returns the character mapped to the ordinal number *i*. For example, the following function call returns a value of 'A': `CHR(65)` A run-time error message is issued if *i* is less than 0 or greater than 127 and run-time checking is enabled.
ORD(*c*)	Returns the ordinal number for the character *c*. For example, the following function call returns a value of 65: `ORD('A')`
PRED(*c*)	Returns the character that is the predecessor of *c* in type CHAR ordering. For example, assuming the variable ch has a value of 'B', the following function call returns 'A': `PRED(ch)` If run-time checking is enabled and *c* is the first element in the type, a run-time error message is issued.
SUCC(*c*)	Returns the character that succeeds *c* in the type CHAR ordering. For example, assuming the variable ch has a value of 'A', the following function call returns 'B': `SUCC(ch)` If run-time checking is enabled and *c* is the last element in the type, a run-time error message is issued.

5.7 ENUMERATED TYPES

An enumerated type contains constants defined by you. This type can either be defined explictly in a TYPE definition or implicitly in a VAR declaration (see section 4, Program Organization). The TYPE definition assigns a name to the enumerated type, while the VAR declaration does not.

An enumerated type is defined as follows:

```
enumerated-type = "(" id-list ")" .
```

The constants receive ordinal numbers, beginning with 0, in the order they are listed.

Example:

```
TYPE directions = (east, west, north, south);
VAR compass: directions;
```

In this example, east receives the ordinal number 0, west 1, north 2, and south 3. The variable compass is declared to be of type directions and can be assigned any of the enumerated constants. The relational operators and the ORD, PRED, and SUCC functions can refer to these constants as follows:

```
compass = north
ORD(south)
PRED(north)
SUCC(east)
```

The Pascal compiler checks assignments made to variables of enumerated types. If a program attempts to assign a value of the wrong type to such a variable, the compiler flags an error.

5.8 SUBRANGE TYPES

As the name implies, a subrange type is composed of constants that are a subrange of any previously defined scalar type except REAL. You designate these constants. As with enumerated types, a new subrange type can either be defined explicitly in a TYPE definition or implicitly in a VAR declaration (see section 4, Program Organization).

A subrange type is defined as follows:

```
subrange-type = constant ".." constant .
```

The first constant specifies the first element in the type and the second constant specifies the last. All elements that lie between the named constants in the ordering of the previously defined type are included in the new type.

Example:

```
TYPE degrees = -45..120;
VAR temperature: degrees;
```

In the example, the new type degrees is defined as a subset of integers from -45 to 120. If a value outside the subrange is assigned to the variable temperature and run-time checking is enabled, the program aborts with an error message.

The relational operators and the PRED and SUCC functions can refer to variables or expressions of subrange types.

Example:

```
SUCC(temperature)
```

5.9 ARRAY TYPE

An array permits you to treat multiple elements of the same type as a single structure, using a single name. The format for an array declaration in a TYPE definition is as follows:

```
array-type = "array" "[" index-type { "," index-type } "]"
             "of" component-type .

index-type = ordinal-type .

component-type = type-denoter .
```

The index-type specifies the ordinal, or subscript, data type by which individual elements are accessed and gives the size of the array. (The maximum array size is 2^{24}-1 words, which is 16,777,215 decimal.) If you specify two or more index-types, the array is multidimensional. The index-type is most frequently a subrange of integers. However, it could be any scalar type, or a subrange of any scalar type, except REAL.

The component-type specifies the type of the data held in the array. If a program attempts to enter data of a different type, an error message is issued.

Example:

```
TYPE  token = ARRAY [1..10] OF CHAR;
VAR    x: token;
      ch: CHAR;
       i: INTEGER;
```

In the example, an array type named token is defined to be 10 elements of type CHAR. The variable x is declared to be of type token. To access one of the elements, the form x[i] is used, where i is an integer from 1 through 10 inclusive. After the following assignment statement (described in section 7, WITH and VIEWING Statements), element i in the array x contains the value of variable ch:

```
x[i] := ch;
```

The following example uses an enumerated type as an index:

```
TYPE days = (sunday, monday, tuesday, wednesday, thursday, friday,
             saturday);
     pettycash = ARRAY [days] OF REAL;
VAR balance: pettycash;
```

An element in the array defined in the example is accessed as follows:

```
balance[monday]
```

Internally, each element of any unpacked array of a predefined type uses one 64-bit word of memory in the Cray computer system. Returning to the first example, figure 5-1 represents how array x is stored when its first five elements contain the character A and the second five contain the character B.

5.9.1 PACKED ARRAYS

Rather than storing one array element in each 64-bit word, memory can be used more efficiently if the array is defined as a *packed* array. The same array depicted in figure 5-1 can be defined as follows:

```
TYPE  token = PACKED ARRAY [1..10] OF CHAR;
VAR    x: token;
      ch: CHAR;
```

If the first five elements in the array are set to the character A and the last five are set to B, the packed array is then represented internally as in figure 5-2.

The next element in a packed structure begins on the next available bit unless the element is 64 bits or longer. Elements of 64 bits or more are word-aligned.

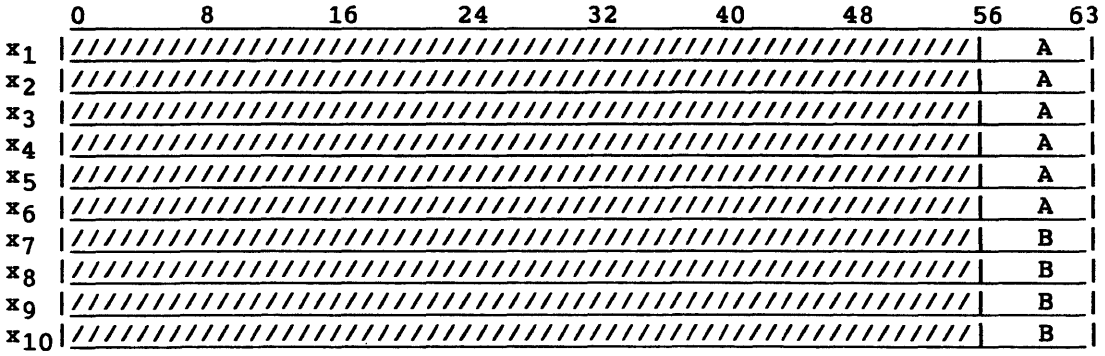

Figure 5-1. Internal Representation of an Unpacked Array

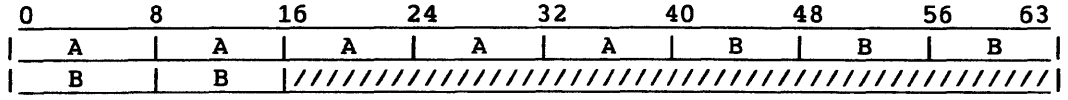

Figure 5-2. Internal Representation of a Packed Array

NOTE

While a packed array is smaller in size, more complex code may be generated to access its elements. This could slow program execution and increase the size of the code necessary to process the packed data. Also, Pascal does not vectorize operations on packed arrays. Making the decision of whether or not to pack a given array may require experimentation to determine if the reduction in array memory space offsets the increase in total code size and execution time. (The summary message at the end of the program listing gives the amount of memory used by the program.) In general, an unpacked structure is more efficient for an array that is small and frequently accessed. A large array that is seldom accessed may be more efficiently stored in a packed structure.

The contents of an unpacked array can be copied into a packed array and vice versa by predefined procedures PACK and UNPACK, respectively. (See appendix B, Predefined Functions and Procedures, for descriptions of all predefined functions and procedures.)

Function	Description
PACK(*u,i,p*)	Takes the elements from the unpacked array (*u*), beginning at the specified index position (*i*), and copies them into the packed array (*p*), beginning at the first position.
UNPACK(*p,u,i*)	Takes the elements from the packed array (*p*), beginning at the first position, and copies them into the unpacked array (*u*), beginning at the specified index position (*i*).

5.9.2 MULTIDIMENSIONAL ARRAYS

Arrays of more than one dimension are defined by adding more than one index-type to the declaration. The following example defines a two-dimensional array:

```
TYPE twodimen = ARRAY [1..10,1..10] OF INTEGER;
VAR  matrix: twodimen;
     i, j:   1..10;
```

The two-dimensional array can then be accessed through the use of variables i and j or by integer constants in the range of 1 through 10.

Examples:

```
matrix[i,j]
matrix[5,9]
```

The PACKED declaration applies only to one level of an array unless it is repeated to apply to a second level. For instance, only the 80-element array is a packed array in the following declaration:

```
VAR deck : ARRAY [1..4096] OF PACKED ARRAY [1..80] OF CHAR;
    i, j : INTEGER;
```

The following declarations are alternate methods of packing both arrays:

```
VAR deck : PACKED ARRAY [1..4096] OF PACKED ARRAY [1..80] OF CHAR;
VAR deck : PACKED ARRAY [1..4096, 1..80] OF CHAR;
```

Elements in the array deck are accessed in either of the following ways:

```
deck[i,j]
deck[i][j]
```

Similarly, the following declaration sets up an array of 4 words. Each of the words contains a 2-bit packed array in its leftmost bits:

```
VAR small : ARRAY [1..4] OF PACKED ARRAY [1..2] OF BOOLEAN;
```

5.9.3 STRINGS

A string is a packed array of characters. The individual characters are taken as a whole and treated as a single object. A string is delimited by apostrophes. An apostrophe is represented by two apostrophes when appearing in a string.

Examples:

```
'string'
'don''t'
```

The first example might have been defined as follows:

```
TYPE words = PACKED ARRAY [1..6] of CHAR;
VAR  x: words;
.
.
.
x := 'string';
```

All strings can be accessed by reference to the variable (x in this case) rather than to the individual array elements.

When a value is assigned to a string variable, the number of characters assigned to the string variable must match the declared size of the variable. If a READ statement reads a string that contains less characters than the number of characters declared for a string variable, the number of blanks equal to the difference between the declared size of the string variable and the size of the string are added to the right of the last character read. (See section 10, Input and Output, for a description of the READ statement and reading a string.)

Example:

```
VAR word: PACKED ARRAY [1..8] OF CHAR;
.
.
.
word := 'Cray    ';
```

Figure 5-3 illustrates the internal representation of this example.

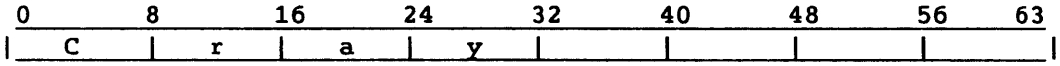

Figure 5-3. Internal Representation of a String

Strings of the same type can be compared using the relational operators. The order is based on the collating sequence of the elements in the strings. (Appendix A, Character Set, gives the collating sequence for type CHAR.) Table 5-4 lists the relational operations for use on strings. In table 5-4, string variable x has a value of 'xxx', and string variable y has a value of 'yyy'.

Table 5-4. String Operations

Operation	Result	Description
x = y	FALSE	Tests for x equal to y
x > y	FALSE	Tests for x greater than y
x < y	TRUE	Tests for x less than y
x >= y	FALSE	Tests for x greater than or equal to y
x <= y	TRUE	Tests for x less than or equal to y
x <> y	TRUE	Tests for x not equal to y

5.10 TYPE ALFA

Type ALFA is a predefined string. ALFA is equivalent to the following definition:

```
TYPE ALFA = PACKED ARRAY [1..8] OF CHAR;
```

Values are assigned to variables of type ALFA just as to any other packed array of characters.

Example:

```
VAR x: ALFA;
.
.
.
x := 'Cray    ';
```

Operands of type ALFA can be used in the operations described earlier in table 5-4. Both operands must be of the same type, however. If an operand of type ALFA is compared to an operand of another string type, even if that type has the same characteristics as ALFA, an error message is generated. Variables x and y in the following example, for instance, are not equivalent:

```
VAR x : ALFA;
    y : PACKED ARRAY [1..8] OF CHAR;
```

Type ALFA is a CRI extension to the ISO Level 1 Pascal standard.

5.11 RECORD TYPES

The record goes a step beyond an array by permitting you to treat multiple elements of different types as a single structure with a single name. The parts that make up a record, called *fields*, can each be of any scalar, structured, or pointer type.

The format of a record definition is as follows:

```
record-type = "record" field-list "end" .

field-list = [ ( ( fixed-part [ ";" variant-part ] ) | variant-part )
             [ ";" ] ] .

fixed-part = record-section { ";" record-section } .

variant-part = "case" variant-selector "of" variant { ";" variant } .

variant-selector = [ tag-field ":" ] tag-type .

tag-field = identifier .

tag-type = ordinal-type-id .

variant = case-constant-list ":" "(" field-list ")" .
```

The maximum size of a record is $2^{24}-1$ words (16,777,215 decimal).

Example:

```
TYPE personnel = RECORD
                   name: PACKED ARRAY [1..18] OF CHAR;
                   empnumber, age: INTEGER;
                   sex: (male, female);
                   salary: REAL;
                   vested: BOOLEAN
                 END;
```

In this example, the record named personnel has six fields. The fields number and age are both of type INTEGER. The field sex is an enumerated type containing the two constants listed.

You can also define more complex record structures. For example, a record can be defined within a record:

```
TYPE personnel = RECORD
                   name: PACKED ARRAY [1..18] OF CHAR;
                   address: RECORD
                              street, city:  PACKED ARRAY [1..20] OF
                                             CHAR;
                              state: PACKED ARRAY [1..2] OF CHAR;
                              zip: INTEGER
                            END;
                    .
                    .
                    .
                   vested: BOOLEAN
                 END;
```

5.11.1 VARIANT FIELDS

A record can be defined with *variant* fields. A variant field may or may not be selected, depending on the result of a condition specified in a CASE clause. (See section 8, Assignment Statement and Program Control Statements, for a description of the CASE statement.) Continuing with the previous example of a record, the field vested could be set up with variant fields as follows:

```
TYPE personnel = RECORD
                    .
                    .
                    .
                   CASE vested: BOOLEAN OF
                    TRUE: (amount, percentage: REAL);
                    FALSE: ()
                 END;
```

In the previous example, the CASE clause tests the value of the field vested (called the *tag* field). The possible values of the tag field vested, which is of type BOOLEAN, are TRUE and FALSE. If the value is TRUE, the field variables amount and percentage are declared as type REAL. If vested has a value of FALSE, no variables are declared. Thus, if an array of the above records was defined, some records might contain the variables amount and percentage and some might not.

The data type for a tag field must be a scalar type other than REAL.

5.11.2 PACKED RECORDS

A packed record is declared in the same manner as an unpacked record, with the addition of the reserved word PACKED in front of the word RECORD.

Example:

```
TYPE personnel = PACKED RECORD
.
.
.
```

In a packed record, most data items are allocated the exact number of bits required. The exceptions to this rule are as follows:

- Items longer than 1 word (64 bits) are begun on a word boundary rather than in the middle of a word.
- A data item of 64 bits or less is never split across a word boundary.
- The fields in the last word of a multiword packed record are right-justified rather than left-justified.

Since an unpacked record is given at least 1 word for each data item, the following record uses 6 fewer words than its unpacked equivalent:

```
TYPE savespace = PACKED RECORD
                   i, j, k: CHAR;
                   bool1, bool2: BOOLEAN;
                   t: I24;
                   status: (first, middle, last)
                 END;
```

Figure 5-4 shows how a variable of type savespace is stored.

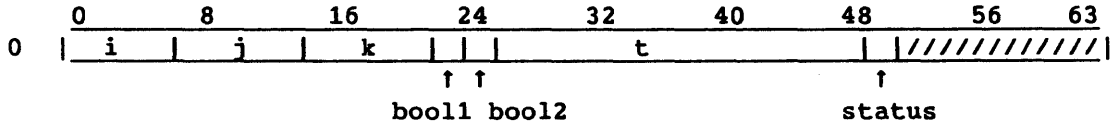

Figure 5-4. Internal Representation of a Packed Record

Figure 5-5 shows the storage for the same record declared as an unpacked structure.

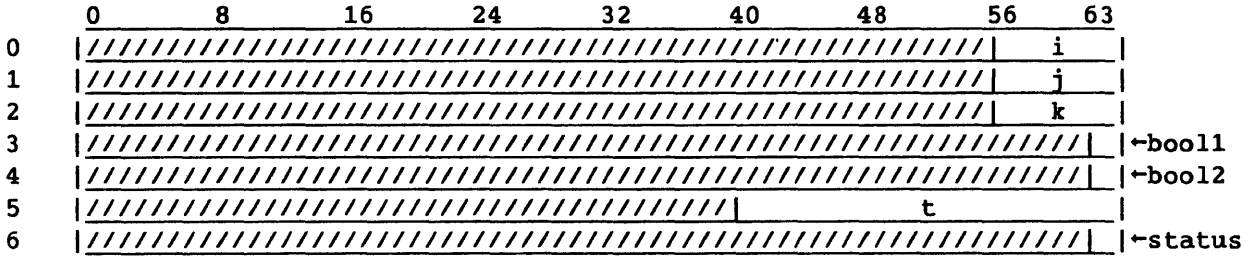

Figure 5-5. Internal Representation of an Unpacked Record

A significant amount of memory may not always be saved, however, as the following records illustrate:

```
TYPE twowdrec = PACKED RECORD
                  i, j, k: I24
                END;
TYPE biggerec = PACKED RECORD
                  x, y: I24;
                  z1, z2: twowdrec
                END;
```

The packed record twowdrec occupies 2 words, with i and j in the first word and k in the second. However, the packed record biggerec occupies 5 words. Since z1 and z2 are longer than 1 word, they both begin on word boundaries. Thus, x and y occupy the first word, z1.i and z1.j the second, z1.k the third, z2.i and z2.j the fourth, and z2.k the fifth.

NOTE

As with a packed array, a packed record may require the generation of more complex code to access its elements. This could slow program execution and increase the code size necessary to access the packed data. Making the decision of whether or not to pack a given record may require experimentation to determine if the reduction in record memory space offsets the increase in total code size and execution time. (The summary message at the end of the program listing gives the amount of memory used by the program.) In general, an unpacked structure is more efficient for a record that is small and frequently accessed. A large record that is seldom accessed may be more efficiently stored in a packed structure.

5.11.3 ACCESSING RECORD FIELDS

To access fields of a record, variables are declared and associated with the record type.

Example:

```
VAR worker: personnel;
```

Often an array of records is desirable, in which case the variable is defined as an array.

Example:

```
CONST totemployees = 637;
TYPE personnel = RECORD
                   .
                   .
                   .
                 END;
VAR employee: ARRAY [1..totemployees] OF personnel;
```

In either case, a record field is accessed using the following form:

```
record-variable "." field
```

Examples:

```
worker.empnumber
employee[1].salary
```

The data type POINTER (described in this section) and the WITH structure (described in section 7, WITH and VIEWING Statements) are also frequently used in accessing record fields.

5.12 SET TYPES

A set type contains elements specified by you. Unlike an array or a record, a set allows you to treat the elements as a group rather than as individual entities. A set type is defined as follows:

```
set-type = "set" "of" base-type .
base-type = ordinal-type .
ordinal-type = new-ordinal-type | ordinal-type-id .
```

The base type of a set must be one of the following:

- An enumerated type of as many as 128 elements
- A subrange of type INTEGER or type BOOLEAN with a maximum range of 0 through 127
- A subrange of type CHAR

Examples:

```
TYPE all    = SET OF ' '..'z';
     caps   = SET OF 'A'..'Z';
     days   = SET OF (sunday, monday, tuesday, wednesday, thursday,
                      friday, saturday);
```

Set variables are declared in the VAR declaration.

Examples:

```
VAR       w: SET OF 'a'..'z';
        x, y: caps;
```

The assignment statement assigns values to set variables. (Section 8, Assignment Statement and Program Control Statements, describes the assignment statement.)

Examples:

```
x := ['A'..'O'];
y := ['L'..'Z'];
z := ['W'];
```

Table 5-5 describes the operations for sets. The variables x and y are assumed to have been declared and assigned values as in the previous examples.

Table 5-5. Set Operations

Operation	Result	Description
x + y	'A'..'Z',	Creates a new set that is the union of the two sets specified as operands. All elements in either x or y are included in the new set.
x - y	'A'..'K'	Creates a new set that is the symmetric difference of the two sets specified as operands. Only those elements in set x that are not also elements in set y are included in the new set.
x * y	'L'..'O'	Creates a new set that is the intersection of the two sets specified as operands. All elements that are in both sets are included in the new set.
x = y	FALSE	Tests for set equality. The result is TRUE if both sets contain the same elements.
x <> y	TRUE	Tests for set inequality. The result is TRUE if the two sets do not contain the same elements.
x <= y	FALSE	Tests for set inclusion. The result is TRUE if every element in x is also an element in y.
x >= y	FALSE	Tests for set inclusion. The result is TRUE if every element in y is also an element in x.
'W' IN y	TRUE	Tests for set membership. The result is TRUE if the element specified by the first operand is a member of the set specified by the second operand.

5.13 FILE TYPES

A file contains elements of the same type that are accessed sequentially. A file type is declared as follows:

```
file-type = "file" "of" component-type .
```

The component type cannot be FILE. Any other data type in Pascal can be the component type, although restrictions may apply to some. For instance, a file of pointers may be invalid if the file is saved and read back into memory at a later time. The component type can be array if the array type has been defined in the declarations section of the block.

Example:

```
TYPE stats = FILE OF INTEGER;
     names = FILE OF CHAR;
```

Records and arrays can have files as components. The following example creates an array of six files. Unless a file is listed as a PROGRAM heading parameter, Pascal creates an internal name for the file.

```
VAR filegroup : ARRAY[0..5] OF FILE OF INTEGER;
     WRITE (filegroup[i],i); (* Write file_i with i *)
```

A file is accessed sequentially through a *buffer variable,* which permits access to the file elements one at a time. The position of the buffer variable in the file may be thought of as a *window* through which the data becomes accessible. The window advances through the file element by element to allow data to be read from, or written to, each file position. When an element of data is read from the window position in a file, it is entered into the buffer variable; from there it can be assigned to an element in an array, for instance. For a write operation, the data is first assigned to the buffer variable and then written to the window position at the end of the file.

A buffer variable of the same type as the file is created implicitly for each file declared. The buffer variable is referenced using the following form:

```
file-name "^"
```

The following predefined functions and procedures operate on files. *fn* represents a file name and *ldn* represents a local dataset name.

Function or Procedure	Description
EOF(*fn*)	Returns a value of TRUE if an end-of-file condition exists for *fn*. The end-of-file condition exists if the window is past the last element in the file. The predefined file INPUT is the default if *fn* is not specified.
EOLN(*fn*)	Returns a value of TRUE if the window is currently positioned on an end-of-line character. The predefined file INPUT is the default if *fn* is not specified.
RESET(*fn*)	Moves the window to the first element in *fn* and, unless the file is empty, sets the end-of-file condition to FALSE. Any file that is to be read, except the standard file INPUT, must be reset first.
REWRITE(*fn*)	Clears *fn* and sets the end-of-file condition to TRUE. Any file that is to be written, except the standard file OUTPUT, must be cleared first.
GET(*fn*)	Moves the window to next element of *fn* and assigns the value of that element to the buffer variable. If the next element is the end-of-file, the buffer variable is undefined, and the end-of-file condition is set to TRUE.
PUT(*fn*)	Writes the contents of the buffer variable to the window position at the end of file *fn*
CONNECT(*fn,ldn*)	Associates the Pascal file *fn* with the COS local dataset *ldn*

Section 10, Input and Output, describes these predefined functions and procedures, the standard text files INPUT and OUTPUT, and the predefined procedures READ, WRITE, READLN, and WRITELN in more detail.

The predefined function EOLN can only be used with the standard file type TEXT. A text file is composed of lines of printable characters. Each line ends with an end-of-line indicator.

If a text file other than the standard files INPUT and OUTPUT is used in a program, it must be named in a VAR declaration and be included as a parameter in the program heading.

Example:

```
PROGRAM test (infile, outfile, OUTPUT);
  .
  .
  .
  VAR infile, outfile: TEXT;
```

NOTE

A line in a text file cannot exceed 140 characters. This limit is an implementation restriction to the ISO Level 1 Pascal standard, which does not limit the size of a line in a text file.

Cray Pascal supports datasets in the blocked format for COS. Blocked datasets are described in the COS Version 1 Reference Manual. Unblocked datasets are supported only through FORTRAN library routines described in the Programmer's Library Reference Manual. When using these FORTRAN routines on unblocked datasets, you are responsible for buffer management. (Section 8, Assignment Statement and Program Control Statements, describes how to call FORTRAN library routines.)

Cray Pascal supports standard UNICOS files, which are described in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual. If blocked files are specified, you are responsible for all buffer management. The UNICOS File Formats and Special Files Reference Manual describes the format for blocked files.

Files cannot be compared using the relational operators.

5.14 POINTER TYPE

A pointer value contains the address of a value rather than the value itself. The pointer type gives Pascal the facility for creating dynamically allocated variables, as opposed to variables allocated during the compile pass. (Section 11, Dynamic Allocation, describes dynamic allocation.)

A pointer variable is declared as being of the pointer type. The pointer type is defined as follows:

```
pointer-type = new-pointer-type | pointer-type-id .

new-pointer-type = "^" domain-type .

domain-type = type-id .

pointer-type-id = type-id .
```

Examples:

```
TYPE pointer = ^INTEGER;
VAR  p: pointer;

TYPE ptr  = ^node;
     node = RECORD
              data      : INTEGER;
              next_node : ptr
            END;
```

In the first example, pointer is declared as a pointer type *bound* to type INTEGER. Thus, any variables of type pointer can be used only with integers. If the program attempts to use such variables with any other data type, an error message is generated. The identifier p is declared as a variable of type pointer. p points to a dynamic variable when one is created. The p^ form accesses the value of that dynamic variable.

In the second example, ptr is declared as a pointer type bound to the user-defined type node. The type node can be defined after it is referenced in the declaration of ptr. Such a forward reference is valid, but node must be defined in the same TYPE declaration in which it is referenced. If the definition of node were separated from the definition of ptr by another kind of declaration, such as a VAR declaration, the forward reference to node in the ptr definition would be invalid.

A pointer contains the address of the value to which it points, stored right-justified in a 64-bit word. Figures 5-6 and 5-7 show the internal representation for the pointer p in the first example.

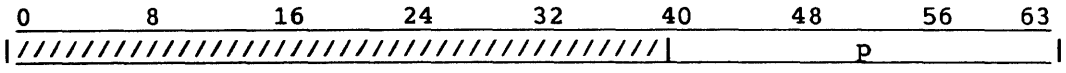

Figure 5-6. Internal Representation of a Pointer (CRAY X-MP and CRAY-1 Computer Systems)

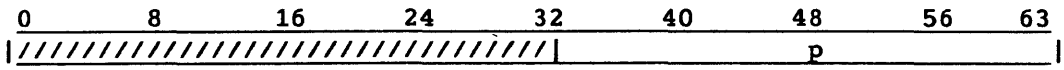

Figure 5-7. Internal Representation of a Pointer (CRAY-2 Computer System)

Except for the special value NIL, pointers can be assigned values only from other pointers or through the predefined functions NEW and DISPOSE (see section 11, Dynamic Allocation). A pointer containing the value NIL points to nothing. Pointers can only be compared for equality (=) and inequality (<>).

The following procedures and functions accept parameters or return results of type pointer.

Function or Procedure	Description
NEW(x) (procedure)	Defines a dynamically allocated variable pointed to by x. NEW is passed a VAR parameter that is a pointer variable. A dynamic variable of the type pointed to by x is allocated, and x is set so that it points to the dynamic variable. Thus x^ becomes defined.
DISPOSE(x) (procedure)	Deallocates the dynamically allocated variable pointed to by x. DISPOSE is called by an expression of type pointer. The dynamic variable pointed to is deallocated; x^ thus becomes undefined.
LOC(x) (function)	Returns the address of x; this address is type-compatible with pointers. Use of the LOC function inhibits assignment of any user variables to the B and T registers for the entire compile unit.

Section 11, Dynamic Allocation, discusses NEW and DISPOSE more fully.

The LOC function returns a pointer for the specified variable; this pointer is assignment-compatible with all types of pointers. LOC uses one VAR parameter, which cannot be an element of a packed structure or tagfield.

LOC escapes strong data typing. Strong typing is fundamental to Pascal; circumventing it tends to make programs more difficult to maintain.

Because LOC can generate pointers to data on the stack or in static or common storage, allocation of variables to B or T registers is disabled in programs or modules that use LOC. This precaution prevents the creation of binary programs that load variable values from B or T registers after their values have been modified by pointers set by the LOC function.

Because the pointer returned by the LOC function fails full run-time pointer checking, use one of the following methods in programs or modules that use the LOC function:

- Disable pointer checking with (*#RP- *).
- Enable null pointer checking with (*#RPN *).

The LOC function can acquire the address of its own argument in the following way:

```
integer_address_of_the_argument := ORD(LOC(the_argument));
```

The LOC function, and the capability of taking the ORD of a pointer, are CRI extensions to the ISO Level 1 Pascal standard.

For manipulating a variable var1 of type typea as if it were of type typeb, a pointer p1 can be declared to be of type ^typeb and the following code can be executed:

```
(*#RPN turn off pointer checking *)
p1 := LOC(var1);
(* manipulate p1^ as required *)
(*#RP+ turn pointer checking back on *)
```

6. ARRAY PROCESSING

Cray Pascal supports a set of explicit array processing constructs. These language features allow operations on arrays to be described with a notation that is simple and intuitive. Since array processing operations are easily vectorized, the execution time of a program is decreased. The types of array processing constructs are as follows:

- Array expressions constructed with binary and unary operators
- Reduction functions
- Constructed arrays
- Array merges
- Relational operators

NOTE

Array bounds range checking, which is enabled by default, inhibits the vectorization of array processing operations. Operations which use PACKED arrays are also not vectorized.

6.1 ARRAY EXPRESSIONS - BINARY AND UNARY OPERATORS

The binary ("+", "-", "*", "/", DIV, MOD, AND, and OR) and unary ("+", "-", and NOT) operators operate on arrays as well as on simple values. If one operand of a binary operator is an array and the other is a simple value, the simple value is expanded into an array of the same shape as the array operand.

If both operands of a binary operator are arrays, they must be of the same shape; that is, both arrays must possess the same number of dimensions, and corresponding dimensions must be the same size (although they need not have the same upper and lower bounds).

For example, suppose the variables a, b, c, and d are declared with the following:

```
VAR
     a, b: ARRAY [1..10, 1..5] OF REAL;
     c: ARRAY [1..5, 1..10] OF REAL;
     d: ARRAY [3..7, 11..20] OF REAL;
```

The following assignment is valid, because array a has the same shape as array b:

```
a := b;
```

The following assignment is invalid, because array a does not have the same shape as array c. A compiler error is generated as a result.

```
a := c;
```

The following assignment is valid. Although the upper and lower bounds of arrays c and d are different, the shape of the arrays is the same:

```
c := d;
```

Suppose the variables a, b, and c are declared with the following:

```
VAR
     a, b: ARRAY [1..10, 1..10] OF REAL;
     c: ARRAY [0..9, 0..9] OF INTEGER;
```

The following statement adds array a to array b on an element-by-element basis and assigns the resulting array to a:

```
a := a + b;
```

The following statement multiplies array a and array c (whose elements are implicitly converted from integers to real numbers) on an element-by-element basis and assigns the resulting array to a:

```
a := a * c;
```

The following statement doubles each element of array a:

```
a := a * 2.0;
```

Arrays and array expressions may be used as arguments to the following predefined functions:

ABS	LN
ARCCOS	LOG
ARCSIN	ORD
ARCTAN	POP

BAND	PRED
BNOT	SIN
BOR	SINH
BXOR	SQR
CHR	SQRT
COS	SUCC
COSH	TAN
EXP	TANH

NOTE

ABS, ARCCOS, ARCSIN, ARCTAN, BAND, BNOT, BOR, BXOR, CHR, COS, COSH, EXP, LN, LOG, ORD, POP, PRED, SIN, SINH, SQR, SQRT, SUCC, TAN, and TANH are the only procedures or functions (other than the array expression reduction functions described in this section) to which an array expression can be passed as an actual parameter.

The result of an array expression can be assigned to any array variable of the same size. If a simple value is assigned to an array, the simple value is assigned to every element in an array. For example:

```
a := 0.0;
```

assigns 0.0 to every element in array a.

Expressions that operate on entire arrays are a CRI extension to the ISO Level 1 Pascal standard.

6.2 REDUCTION FUNCTIONS

The standard reduction functions (ALL, ANY, MAXVAL, MINVAL, PRODUCT, and SUM) reduce array expressions into simple values. Each of the reduction functions takes one argument, which can be an array of any shape, and combines all of the argument's elements into a simple value. Table 6-1 lists and briefly describes the Cray Pascal reduction functions.

Table 6-1. Pascal Reduction Functions

Function	Argument Type	Result Type	Description
ALL	Boolean	Boolean	Returns TRUE if all of the elements are TRUE
ANY	Boolean	Boolean	Returns TRUE if any one of the elements is TRUE
MAXVAL	Any scalar type	Same as argument	Returns the element with the highest value
MINVAL	Any scalar type	Same as argument	Returns the element with the lowest value
PRODUCT	Real or integer	Same as argument	Calculates the product of all of the elements
SUM	Real or integer	Same as argument	Calculates the sum of all of the elements

Reduction functions (ALL, ANY, MAXVAL, MINVAL, PRODUCT, and SUM) are CRI extensions to the ISO Level 1 Pascal standard.

6.3 CONSTRUCTED ARRAYS

Arrays can be constructed from other arrays in several ways. Each method builds a new array from the elements of another. A constructed array can be used as an operand in an array expression or as the target variable of an assignment statement. Constructed arrays include the following constructs that are described later in this subsection:

- Array-valued subscripts
- Slice index specification
- Array-valued field and pointer accesses

6.3.1 ARRAY-VALUED SUBSCRIPTS

An array can be subscripted by an array expression whose element type is compatible with the index type of the array. The result has the same shape as the index expression and is constructed from the elements of the original array. The order in which array elements are accessed is not defined.

Examples:

```
VAR
   a: ARRAY [1..10] OF REAL;
   b: ARRAY [1..5] of 1..10;
   c: ARRAY [1..2, 1..2] OF INTEGER;
```

a [b] is of the type ARRAY [1..5] OF REAL and contains the following:

```
a [b [1]]
a [b [2]]
a [b [3]]
a [b [4]]
a [b [5]].
```

a [c] is of type ARRAY [1..2, 1..2] OF REAL and contains the following:

```
a [c [1,1]]
a [c [1,2]]
a [c [2,1]]
a [c [2,2]]
```

6.3.2 SLICE INDEX SPECIFICATION

An array can be constructed from some of the elements of another array using a slice index specification. The syntax of the slice index specification is as follows:

```
slice = ".." | expression ".." expression ["..." expression].
```

A slice can be specified in three forms. The first form is specified as follows:

```
slice = ".."
```

Form 1 selects all of the elements in the array. Form 2 is specified as follows:

```
slice = beginning ".." ending
```

Form 2 selects a sequence of elements, beginning at the element indexed by the first expression and ending at the element indexed by the second. Form 3 is specifed as follows:

```
slice = beginning ".." ending ".." stride
```

Form 3 includes a stride value that specifies the distance between selected elements. The first and second expressions must be compatible with the index type of the array; the third expression must be compatible with the type of INTEGER and must be nonzero.

Examples:

```
VAR
   a, b: ARRAY [1..10] OF REAL;
   c: ARRAY [1..5, 1..5] OF REAL;
```

The statement (a [1..5] := 0.0) makes the following assignments:

```
0.0 to a [1]
0.0 to a [2]
0.0 to a [3]
0.0 to a [4]
0.0 to a [5]
```

The statement (a [1..9..2] := b [1..5]) makes the following assignments:

```
b [1] to a [1]
b [2] to a [3]
b [3] to a [5]
b [4] to a [7]
b [5] to a [9].
```

The following statement assigns the third column of array c to the initial elements of array a:

```
a [1..5] := c [.., 3]
```

The following statement reverses the elements in array a:

```
a [1..10] := a [10..1..-1]
```

When slice indexes are specified, it is often impossible to determine the size of the resulting array until run time. Compatibility of operands cannot be fully checked when the program is compiled. The RL option controls the run-time compatibility check. If the RL option is on (RL+), code is generated to ensure that operands have the same shapes.

Slice index specification is a CRI extension to the ISO Level 1 Pascal standard.

6.3.3 ARRAY-VALUED FIELD AND POINTER ACCESSES

An array can be constructed from another array by specifying a field or pointer access with an array as the base variable. The result is an array of the same shape as the base variable.

Examples:

```
VAR
   a: ARRAY [1..10] OF RECORD
                          f1: INTEGER
                       END;
   b: ARRAY [1..10] OF INTEGER;
   c: ARRAY [1..5] OF ^ RECORD
                          f2: INTEGER
                       END;
```

The statement b := a.f1; is equivalent to the following:

```
FOR i := 1 TO 10 DO
   b[i] := a[i].f1;
```

The statement (a [1..5].f1 := c^.f2) is equivalent to

```
FOR i := 1 TO 5 DO
   a[i] := c[i]^.f2
```

Array-valued field and pointer accesses are a CRI extension to the ISO Level 1 Pascal standard.

6.4 ARRAY MERGES

A conditional expression has the following form:

"IF" subexp "THEN" subexp "ELSE" subexp.

When the first subexpression of the conditional expression is an array expression, the conditional expression denotes an array merge operation.

The second and third subexpressions must be simple values or array expressions whose results are the same shape as the first subexpression. The result of the conditional expression is an array, of the same shape as the first subexpression, whose elements contain the results of the second and third subexpressions.

The source of each element in the result is controlled by the value of the first subexpression. When an element of the result of the first subexpression is TRUE, the element in the result of the merge is the corresponding element in the result of the second subexpression; similarly, when an element of the result of the first subexpression is FALSE, the element in the result of the merge is the corresponding element in the result of the third subexpression.

Example 1:

The following assignment statement replaces all negative elements of a with 0.0:

```
VAR a, b: ARRAY [1..N] OF REAL;
   a := IF a < 0.0 THEN 0.0 ELSE a
```

The assignment statement (a := IF a < 0.0 THEN 0.0 ELSE a) causes each element of array a to be compared with 0.0. If the element being evaluated is less than 0.0, it is replaced with 0.0.

For a description of the way relational operators function within expressions that contain both array variables and scalar variables, see Relational Operators later in this section.

Example 2:

The following statement negates all of the odd elements in array a:

```
VAR a, b: ARRAY [1..n] OF REAL;
   a := IF ODD(a) THEN -a ELSE a
```

Array merges are a CRI extension to the ISO Level 1 Pascal standard.

6.5 RELATIONAL OPERATORS

The following rules determine the meanings of relational operators (<, <=, =, <>, >=, >) in array expressions:

- If both operands of a relational operator are whole arrays (that is, neither operand is the result of an array expression, slice index specification, array-valued subscript, or array-valued base variable), the result is a simple Boolean value containing the outcome of the comparison. For example, the following expression returns a simple Boolean value:

```
s = 'FRED'
```

- Otherwise, the result is a Boolean array of the same shape as the operands whose elements are the outcomes of comparing corresponding elements of the operands. If a simple Boolean result is needed, the result of the relational operator can be reduced with the standard function ALL. For example, the following expression returns a four-element Boolean array:

```
s [1..4] = 'FRED'
```

The difference between s = 'FRED' and s [1..4] = 'FRED' preserves the meanings of relational operators for string expressions in standard Pascal programs while allowing element-by-element comparisons in array expressions.

Example 1:

Example 1 shows how relational operators function within expressions that contain both array variables and scalar variables:

```
VAR a, b: ARRAY [1..n] OF REAL;
   a := IF a < 0.0 THEN 0.0 ELSE a
```

The assignment statement (a := IF a < 0.0 THEN 0.0 ELSE a) causes each element of a to be compared with 0.0. If the element being evaluated is less than 0.0, it is replaced with 0.0.

Example 2:

The following expression returns an n-element Boolean array that contains the results of an element-by-element comparison of array b and array a:

```
VAR a, b: ARRAY [1..n] OF REAL;
   b [..] > a [..]
```

TRUE is returned for every element of array b that is greater than its corresponding element in array a. FALSE is returned for every element of b that is less than or equal to its corresponding element in array a. The assignment statement could also have been specified as follows:

```
a := IF b [..] > a THEN b ELSE a
```

or as

```
a := IF b > a [..] THEN b ELSE a
```

Example 3:

The following expression returns a simple Boolean result when array b and array a are compared.

```
VAR a, b: ARRAY [1..n] OF REAL;
   b > a
```

When array b is compared to array a, a simple Boolean result is returned for the entire array. If the expression b > a is true for every element in array b when it is compared to its corresponding element in array a, the expression is TRUE. If the expression b > a is not true for every element in array b when it is compared to its corresponding element in array a, the expression is FALSE.

Example 4:

Example 4 scales an array of any shape.

```
a := (a - MINVAL (a)) / (MAXVAL (a) - MINVAL (a));
```

Scaling an array reduces every element in an array to a value such that $0 \leq$ array element ≤ 1. If, for example, array a was a 10-element array with 5.0, 6.0, 3.0, 10.0, -1.0, 0.0, 4.0, 3.0, 19.0, and 1.0 as its elements, evaluating the expression produces the following scale values:

Element	Scale Value
a [1] = 5.0	0.3
a [2] = 6.0	0.35
a [3] = 3.0	0.2
a [4] = 10.0	0.55
a [5] = -1.0	0.0
a [6] = 0.0	0.05
a [7] = 4.0	0.25
a [8] = 3.0	0.2
a [9] = 19.0	1.0
a [10] = 1.0	0.1

Example 5:

Example 5 produces the dot product of two vectors (arrays).

```
dot := SUM (a [..] * b [..]);
```

Example 6:

Example 6 performs matrix multiplication.

```
FOR i := i TO m DO
  FOR j := i TO n DO
    o [i,j] := SUM (a [i,..] * b [..,j]);
```

Example 7:

Example 7 performs matrix multiplication more efficiently than example 6.

```
o := 0.0
FOR i := 1 TO m DO
  FOR j := 1 TO n DO
    o [i,..] := o [i,..] + a [i,j] * b[j,..];
```

Example 8:

Example 8 sets up a checkerboard bit map.

```
bits := 0;
bits [1..m..2, 2..n..2] := 1;
bits [2..m..2, 1..n..2] := 1;
```

Example 9:

Example 9 places a limit on the elements of an array.

```
a := IF a > max THEN max ELSE a
```

Example 10:

Example 10 finds all of the primes from 2 to 1,000,000 using the sieve of Erastosthenes.

```
PROGRAM sieve (OUTPUT);
  CONST
    size = 1000000;
  TYPE
    flagarray = ARRAY [2..size] OF BOOLEAN;
  VAR
    flags: flagarray;
    i: INTEGER;
  BEGIN
    flags := TRUE;
    FOR i := 2 TO TRUNC(SQRT(size)) DO
      IF flags[i] THEN
        flags[i+i..size..i] := FALSE;
    WRITELN (' End vectorized Pascal sieve; ',
               SUM (ORD(flags)):0. ' Primes found. ')
  END.
```

Array comparisons are a CRI extension to the ISO Level 1 Pascal standard.

---

7. WITH AND VIEWING STATEMENTS

The WITH and VIEWING statements alter the scopes and types, respectively, of identifiers.

7.1 WITH STATEMENT

The WITH statement offers a shorthand method of referring to record fields. The record structure is named once in a WITH statement. Thereafter, all statements within the scope of the WITH statement can omit the structure name and the period, referring to record fields as if they were simple variables. The form of the WITH statement is as follows:

```
with-statement = "with" record-var-list "do" statement .
```

Example:

```
CONST totemployees = 637;
TYPE personnel = RECORD
                   name: PACKED ARRAY [1..18] OF CHAR;
                   empnumber, age: INTEGER;
                   sex: (male, female);
                   salary: REAL;
                   vested: BOOLEAN
                 END;
VAR employee: ARRAY [1..totemployees] OF personnel;
    i: 1..totemployees;
.
.
.
WITH employee[i] DO
    BEGIN
        name := 'Doe John          ';
        empnumber := 211;
        age := 36;
        sex := male;
        salary := 18191.54;
        vested := TRUE
    END;
```

The values of variables in the record-var-list are determined during the evaluation of the record-var-list, and modifications within the scope of the WITH statement have no effect on those values. For instance, the subscript variable i in the preceding example could be increased by an increment inside the WITH statement, but doing so would not affect the array element accessed.

Using the WITH statement for multiple references to the same variables normally results in more efficient object code.

7.2 VIEWING STATEMENT

The VIEWING statement allows escape from the strict typing rules of standard Pascal. Within the range affected by the VIEWING statement, viewed variables take on new types. The form of the VIEWING statement is:

```
viewing-statement = "viewing" id-list ":" type-id "do" statement.
```

If the statement following the reserved word DO is a compound statement, it is delimited by the reserved words BEGIN and END.

An error message is issued at compile time in the following cases:

- A conformant array appeared in a VIEWING id-list.
- The new type of the variable is larger than the old type. For purposes of use by VIEWING, simple variables of types CHAR, BOOLEAN, INTEGER, I24, REAL, and pointer require one word.

Example:

```
TYPE iarr : ARRAY [1..10] OF INTEGER;
     rarr : ARRAY [1..11] OF REAL;
     rec1 : RECORD
              a : INTEGER;
              c : rarr;
            END;
     rec2 : RECORD
              d : rarr;
              e : INTEGER;
            END;
```

```
VAR
   i,j,k : INTEGER;
   r,s,t : REAL;
   flag  : BOOLEAN;
   ia    : iarr;
   ir    : rarr;
   r1    : rec1;
   r2    : rec2;


BEGIN
   VIEWING i,j,k : REAL DO
    VIEWING flag : REAL DO
      VIEWING ir : REAL DO
       VIEWING r1 : REAL DO
        VIEWING r2 : REAL DO
         BEGIN
           i := r;
           j := s;
           k := t;
           flag := r*s/t;
           ia := flag + 1.0; (* only first word of ia altered *)
           ir := k - 1.0;    (* only first word of ir altered *)
           r1 := j+k+i+ 1.0; (* only first word of r1 altered *)
           r2 := SQRT(i);    (* only first word of r2 altered *)
         END;
   VIEWING r1 : rec2 DO
      r1 := r2;            (* all 12 words of r2 copied to r1 *)
   VIEWING ia : rarr DO    (* error.. rarr is larger than iarr *)
      ia := ir;
   VIEWING ir : iarr DO    (* ok.. iarr is smaller than rarr *)
      ia := ir;            (* only first 10 words copied *)
END;
```

The VIEWING statement is a CRI extension to the ISO Level 1 Pascal standard.

---

8. ASSIGNMENT STATEMENT AND PROGRAM CONTROL STATEMENTS

Statements that manipulate data include the simple assignment statement, conditional branching statements such as the IF and CASE statements, an unconditional branching statement (GOTO), and looping statements such as the REPEAT, FOR, and WHILE statements.

8.1 COMPOUND STATEMENTS

Many of the statements discussed in this section involve the use of the *compound* statement. The WITH and VIEWING statements (see section 7, WITH and VIEWING Statements) may also contain a compound statement. The form of a compound statement is as follows:

```
compound-statement = "begin" statement-sequence "end" .
```

The reserved words BEGIN and END define the scope of a compound statement. Any number of valid Pascal statements can appear within that scope.

The compound statement is treated as a unit and must conform to the rules for other statements. For instance, a semicolon must separate the compound statement from the statement that follows it.

The final statement in the scope of a compound statement need not end in a semicolon, since it is followed by the reserved word END rather than another statement. Placing a semicolon before the END is not, however, an error. The Pascal compiler assumes that such a semicolon separates the preceding statement from a null (empty) statement. For example, the semicolon after the final *statement* in the following compound statement is superfluous but is not an error:

```
statement;
WITH ...
  BEGIN
    statement;
    .
    .
    .
    statement;
  END;
```

8.2 ASSIGNMENT STATEMENT

The assignment statement assigns values to variables. The form of the assignment statement is as follows:

```
assignment-statement = ( var-access | function-id ) ":=" expression .
```

If the variable on the left side of the assignment statement identifies a function, the assignment must occur within the scope of that function. Section 9, Procedures and Functions, describes functions.

Any variable on the left side must be *assignment compatible* with the expression on the right side. If the variable is not assignment compatible, the compiler issues an error message.

A variable and an expression are assignment compatible if any of the following are true:

- The variable and the expression are of the same type (other than type FILE).
- The variable is of type REAL and the expression is of type INTEGER.
- The variable and the expression are sets that are *compatible types,* and all the elements of the expression have been declared as members of the base type of the variable.
- The variable and expression are compatible string types.

Two types are compatible if any of the following are true:

- They are the same type (other than type FILE). Variables of anonymous types are compatible only if they are declared in the same list.
- One is a subrange of the other, or both are subranges of the same type.
- They are set types of compatible base types, and either both are packed or neither is packed.
- They are string types with the same number of elements.
- They are arrays of the same shape. See section 6, Array Processing, for a description of the shape of an array.

The variable can be any of the following:

- A simple variable
- An element in a structured variable (such as an array or record)
- A pointer variable

- A buffer variable (of the form: file-var "^" .)
- A string variable
- A set variable

The expression on the right side of the assignment statement can include any of the following:

- Constant or variable operands
- Operators
- Function identifiers
- Pointers
- Conditional tests

When a function identifier appears on the right side of an assignment statement, that function is invoked.

Examples:

```
ch := 'a';
x := 5.3;
y := SQR(x);
count := count + 1;
bool := x > y;
alphanumerics := letters + ['0'..'9'];
root := p^.link;
table[j,k] := coord.z;
```

Variables of different subranges but of the same underlying type can appear in the same assignment statement as long as the value assigned occurs within both subranges.

Example:

```
VAR var1: 1..18;
    var2: 10..30;
.
.
.
var1 := var2;
```

The preceding assignment statement is valid if var2 only contains elements between 10 and 18. If var2 contains an element outside the range of var1 and run-time checking is enabled, a run-time error message is issued.

The value of one string variable can be assigned to another if both strings are the same length.

Example:

```
VAR string1: PACKED ARRAY [1..8] OF CHAR;
    string2: ALFA;
.
.
.
string1 := 'maniacal';
string2 := string1;
```

8.2.1 CONDITIONAL EXPRESSIONS

A conditional expression permits an IF-THEN-ELSE structure on the right side of an assignment statement and in other places where expressions are valid. The following is a conditional expression if x is a Boolean expression:

```
IF x THEN y ELSE z
```

The types of expressions y and z must be compatible. The result of the conditional expression is y, if x is TRUE; otherwise, the result is z. For example, the following statement writes the string 'Pass' if a is equal to b and 'Fail' if a is not equal to b:

```
WRITELN (IF a=b THEN 'Pass' ELSE 'Fail');
```

The following statement sets MAX to the larger of a and b:

```
MAX := IF a>b THEN a ELSE b
```

Conditional expressions are a CRI extension to the ISO Level 1 Pascal standard.

8.3 IF STATEMENT

The IF statement permits the execution of code within its scope when a specified condition is true. If the optional ELSE clause is present, either of two sections of code are executed depending on whether the condition is true or false. The form of the IF statement is as follows:

```
if-statement = "if" boolean-expression "then" statement [else-part] .
else-part = "else" statement .
```

The statement in both the THEN clause and the ELSE clause can be either a simple or a compound statement. Whenever the statement is compound, it is delimited by BEGIN and END.

Example:

```
IF count < max THEN
  BEGIN
    a[count] := 1;
    count := count + 1
  END
ELSE
  a[count] := 0;
```

In this example, the condition count < max is tested. If it is true, the statements in the THEN clause are executed and the statement in the ELSE clause is not. If the condition is false, the statements in the THEN clause are not executed but the statement in the ELSE clause is. When the ELSE clause is not present and the condition is false, none of the statements within the scope of the IF statement are executed.

Normally, a superfluous semicolon causes a harmless null statement in a Pascal program. In the preceding example, however, a semicolon between the END and the ELSE causes an error. When the statement in the THEN clause is compound and the ELSE clause is present, a semicolon must not follow the reserved word END at the end of the THEN clause.

IF statements can be nested. In nested IF statements, an ELSE clause is always paired with the nearest preceding unpaired THEN clause.

Examples:

```
IF x <= y THEN
  IF (a = b) OR (c = d) THEN
    BEGIN
      .
      .
      .
     IF i = k THEN
       BEGIN
         .
         .
         .
       END
   ELSE ... ;
     .
     .
     .
   END;
```

```
IF a < 0 THEN
  BEGIN
    .
    .
    .
  END
ELSE IF a = 0 THEN
      BEGIN
        .
        .
        .
      END
    ELSE (*If a > 0 *)
      BEGIN
        .
        .
        .
      END;
```

8.4 CASE STATEMENT

As with the IF statement, the CASE statement provides for the conditional execution of a simple or compound statement. Rather than testing only a Boolean expression, however, the CASE statement tests an expression of any scalar type except REAL. If a label in the CASE statement is equal to the value of the expression, the statement following that label is executed.

The form of the CASE statement is as follows:

```
case-statement = "case" case-index "of"
                 case-list-element { ";" case-list-element }
                 [ [ ";" ] "otherwise" [ ":" ] statement ]
                "end" .

case-index = expression .

case-list-element = case-constant-list ":" statement .
```

A CRI extension, the OTHERWISE clause, serves as a label that receives control if none of the other labels match the value of the case-index, or *selector*. The statement in the OTHERWISE clause can be the null statement. If the OTHERWISE clause is not specified and no labels match the value of the selector, and if run-time checking is enabled, a run-time error occurs.

The OTHERWISE clause is a CRI extension to the ISO Level 1 Pascal standard.

The selector is an expression that must resolve to a value of the same type as the labels. More than one label can be used to refer to the same executable statement, as the following example shows:

```
CASE octal_digit OF
  0, 1, 2 : low:= low + 1;
  3, 4, 5 : medium:= medium + 1;
     6, 7 : high:= high + 1;
OTHERWISE : valid:= FALSE
END;
```

The statement following a label may also be a compound statement, as in the following example:

```
CASE ch OF
  'a', 'e', 'i', 'o', 'u':  BEGIN
                              vowelcnt := vowelcnt + 1;
                              previousch := vowel
                            END;
  'y'                       :  CASE previousch OF
                              nonvowel : BEGIN
                                           vowelcnt := vowelcnt + 1;
                                           previousch := vowel
                                         END; (* of nonvowel label *)
                              vowel    : previousch := nonvowel
                            END; (* of inner CASE statement *)
OTHERWISE                   :  previousch := nonvowel
END; (* of outer CASE statement *)
```

8.5 GOTO STATEMENT

The GOTO statement provides an unconditional branch to a statement label. The format of the GOTO statement is as follows:

```
"goto" label .
```

The statement label consists of from 1 to 4 digits (a number from 0 to 9999). The label must be declared in the declarations section of the block in which it appears (see section 4, Program Organization). When it appears in the executable statements section of a program, the label is followed by a colon.

Example:

```
LABEL 10;
.
.
.
GOTO 10;
.
.
.
10 : statement
```

NOTE

The GOTO statement usually produces undesirable results when used to transfer control into a structured statement (for instance, a CASE, IF, REPEAT, WHILE, or FOR statement).

Example:

```
IF x = 0 THEN
  BEGIN
    .
    .
    .
    10: statement
  END;
.
.
.
GOTO 10;
```

8.6 FOR STATEMENT

The FOR statement enables a simple or compound statement to be executed a specified number of times. The format for the FOR statement is as follows:

```
for-statement = "for" control-var ":=" initial-value
                ( "to" | "downto" ) final-value
                [ "by" increment-value ]
                "do" statement .
```

The control variable determines how many times the statement (either simple or compound) following the reserved word DO is executed. The control variable, which must be declared in the declarations section of the block to which it is local, begins with the initial value and increases by an increment value or decreases by a decrement value (or 1 if no BY clause is present) at the end of each iteration. The control variable can be of any scalar type except REAL, and the initial and final values must be assignment-compatible with that type; the increment value, if present, must evaluate to a positive integer.

The BY clause is a Cray Research extension of the ISO Level 1 Pascal standard.

The reserved words TO and DOWNTO increase and decrease the control variable, respectively. The value of the control variable is tested each time the FOR statement executes. If the test reveals that the control variable is greater than the final value (when using TO) or less than the final value (when using DOWNTO), control passes to the statement following the FOR statement.

Pascal evaluates the values controlling iteration in the following order:

1. Initial value
2. Final value
3. Increment value (if present)
4. Assignment to the control variable

NOTE

The value of the control variable cannot be changed within the scope of the FOR statement.

The initial value and final value must be the same type as the control variable. Any scalar type except REAL is valid.

Examples:

```
FOR i := 1 TO 10 BY 2 DO
   x[i] := 0;
```

```
FOR slot := 1 TO linesize DO
  BEGIN
    READ(ch);
    IF EOLN OR EOF THEN
      BEGIN
        FOR rest := linesize DOWNTO slot DO
          line[rest] := ' ';
        GOTO 100
      END
    ELSE
      line[slot] := ch
  END;
```

If the initial value is greater than the final value and TO is specified, or if the initial value is less than the final value and DOWNTO is specified, the loop is not executed. No error message is issued.

The value of the control variable (i, slot, and rest in the preceding examples) is undefined once control exits the FOR statement unless the statement is exited by a GOTO statement. An undefined variable must be reinitialized, either explicitly in an assignment statement or implicitly in another FOR statement, before it can be used again. If the FOR statement is exited through a GOTO statement, as may be the case in the second example, the control variable retains the last value it was assigned.

The following restrictions apply to the control variable in the block in which it is used:

- It cannot be on the left side of an assignment statement.
- It cannot be passed as a VAR parameter in a procedure or function call.
- It cannot appear as a parameter in a READ or READLN statement.
- It cannot be the control variable for more than one nested FOR statement.

Example:

If i is a FOR loop control variable, the following statement generates an error:

```
READ(i);
```

However, the following statement does not generate an error, because i is being used to select a variable and is not a variable itself:

```
READ(line[i]);
```

If the V option is enabled (V+), Pascal attempts to generate code for the FOR statement that exploits the vector processing capabilities of the hardware. See section 13, Vectorization and Optimization, for a description of the types of FOR statements that vectorize.

8.7 REPEAT STATEMENT

The REPEAT statement executes a sequence of statements until a specified condition is true. The form of the REPEAT statement is as follows:

```
repeat-statement = "repeat" statement-sequence
                   "until" boolean-expression .
```

Since the Boolean expression is tested at the end of the REPEAT statement, the statement sequence always executes at least once. As long as the Boolean expression is FALSE, the statement sequence continues to execute. The REPEAT statement is exited when the value of the Boolean expression is TRUE.

Although the statement sequence may be a compound statement, the BEGIN and END keywords need not be specified. The presence of BEGIN and END does not constitute an error.

Example:

```
slot := 1;
REPEAT
  IF EOLN THEN ch := ' '
  ELSE READ(ch);
  line[slot] := ch;
  slot := slot + 1
UNTIL slot > 80;
READLN;
```

8.8 WHILE STATEMENT

The WHILE statement executes a sequence of statements as long as a specified condition is true. The test for the condition comes before the statements are executed. The statements within the scope of the WHILE statement, therefore, do not execute at all if the first test of the condition results in a value of FALSE. The form of the WHILE statement is as follows:

```
while-statement = "while" boolean-expression "do" statement .
```

If the statement following the reserved word DO is a compound statement, it is delimited by the reserved words BEGIN and END.

Example:

```
slot := 1;
WHILE slot <= 80 DO
  BEGIN
    IF EOLN THEN ch := ' '
    ELSE READ(ch);
    line[slot] := ch;
    slot := slot + 1
  END;
READLN;
```

9. PROCEDURES AND FUNCTIONS

Procedures and functions are subprograms within a Pascal program. By using them, you can divide a program into logical components and give it a comprehensible structure.

Procedures and functions can be either internal (coded as part of the program) or external (a routine outside the program, such as a library routine). Although this section deals primarily with internal procedures and functions, external subprograms are also described. Appendix B, Predefined Functions and Procedures, details the predefined Pascal external procedures and functions. The System Library Reference Manual contains information on other external functions and procedures available in the Pascal run-time library. Routines not in the Pascal run-time library are available if compatible parameter passing sequences are used.

Internal procedures and functions are defined in the declarations section of the block to which they are local. They are invoked within the executable statements section of the same block. (See section 4, Program Organization, for a description of the organization of a Pascal program.)

Procedures and functions can be nested within other procedures and functions to a nesting depth of 25.

9.1 PROCEDURES

A procedure is a subroutine that permits a program to be divided into logical parts.

Its form is basically the same as that of a program. Both have a heading, a declarations section, and an executable statements section delimited by the keywords BEGIN and END. The differences between the two are as follows:

- The END that terminates the executable statements section of a procedure is followed by a semicolon rather than a period.
- The forms of the headings are different.

A procedure declaration takes the following form:

```
procedure-dcl =        ( procedure-heading ";" directive ) |
                       ( procedure-identification ";" procedure-block ) |
                       ( procedure-heading ";"
                         [ directive-alt  ";" ]
                         procedure-block ) .

procedure-heading =    "procedure" id [ formal-parm-list ] .

formal-parm-list =     "(" formal-parm-section
                       { ";" formal-parm-section } ")" .

formal-parm-section = value-parm-spec |
                      var-parm-spec |
                      procedural-parm-spec |
                      functional-parm-spec |
                      conformant-array-parm-spec .

directive-alt =        "exported" [ "(" external-name ")" ] .

directive = ( "forward" [ ";" "exported" [ "(" external-name ")" ] ] ) |
            ( "exported" [ "(" external-name ")" ] [ ";" "forward" ] ) |
            "fortran" |
            "external" |
            ( "imported" [ "(" external-name ")" ] ) .
```

Directives for both procedures and functions are described later in this section.

A procedure can use any variables declared in the block to which it is local. For example, if procedure x is declared in procedure y, x can access any variables also declared in procedure y. Variables defined in the main program segment are global; that is, they can be accessed from anywhere in the program block.

Variables declared in the declarations section of a procedure are local to the procedure and are undefined when the procedure is invoked. Variables declared in procedure y, for example, cannot be accessed by the main program that invokes y. The main program and procedure y could both declare variables of the same name. An integer variable i, for instance, could be declared in both. In such a case, the value of the variable i in procedure y is not affected by any assignments to the other variable i in the main program, and vice versa. If procedure x is declared in procedure y and references the variable i, the reference is to the variable i in procedure y rather than in the main program segment.

While using variables defined outside of a procedure is valid, the practice is not always desirable. Assignments to such variables are called *side effects*. Side effects, besides inhibiting readability, may prevent the compiler from optimizing.

Side effects can be avoided by passing variables as parameters. The variables to be passed appear in the statement that invokes the procedure. Each of these parameters (called *actual* parameters) is matched by position to a *formal* parameter in the procedure heading. Parameter passing is described more fully later in this section.

The statement that invokes a procedure has the following form:

```
procedure-statement = procedure-id
                [ ( [ actual-parm-list ] |
                          read-parm-list |
                          readln-parm-list |
                          write-parm-list |
                          writeln-parm-list ) ] .
```

The following is an example of using procedures to divide a program into logical parts:

```
PROGRAM data (INPUT, OUTPUT);
  PROCEDURE inputdata;
  BEGIN
  .
  .
  .
  END; (* inputdata *)

  PROCEDURE processdata;
  BEGIN
  .
  .
  .
  END; (* processdata *)

  PROCEDURE outputdata;
  BEGIN
  .
  .
  .
  END; (* outputdata *)

BEGIN (* data *)
  inputdata;
  processdata;
  outputdata
END.
```

9.2 FUNCTIONS

A function is a subprogram that returns a value. A function is defined in the declarations section of the block to which it is local and is invoked when its name is encountered in the executable statements section of that block.

At least one assignment statement must appear within the scope of the function. That assignment statement gives the function name a value. When the function completes, control returns to the statement from which the function was invoked. The function name in that statement then takes on the value assigned to it in the function.

The form of a function definition is as follows:

```
function-dcl = ( function-heading ";" directive ) |
               ( function-identification ";" function-block ) |
               ( function-heading ";"
               [ directive-alt ";" ]
                 function-block ) .
```

The organization of a function is basically the same as that of a program or a procedure. It has a heading, a declarations section, and an executable statements section delimited by the reserved words BEGIN and END. The directives are the same for both functions and procedures; directives are described later in this section.

The heading of a function is slightly different from both a program heading and a procedure heading:

```
function-heading = "function" id [ formal-parm-list ]
                   ":" result-type .
```

The value returned by a function must be of the result type specified in the function heading. The result type can be a scalar, subrange, or pointer type.

Example:

```
FUNCTION found (symbols: SYMTAB; currentch: CHAR; last: INTEGER):
                BOOLEAN;
(* found determines if the current character is in the array called
   symbols *)
VAR i: INTEGER;
    found1: BOOLEAN;
```

```
BEGIN
  i := 1;
  REPEAT
    found1 := symbols[i] = currentch;
    i := i + 1
  UNTIL (i > last) OR found1;
  found := found1
END; (* of function found *)
```

A function is invoked with a function designator of the following form:

```
function-designator = function-id [ actual-parm-list ] .
```

The appearance of the function name in the following executable statement invokes the function in the preceding example:

```
IF NOT found(symbolarray, ch, lastentry) THEN
  BEGIN
    lastentry := lastentry + 1;
    symbolarray[lastentry] := ch
  END;
```

As with the procedure, any variable appearing in the declarations section of a function is local to that function. Side effects and the loss of compiler optimization may result if a value is assigned to a global variable within a function.

9.3 PARAMETERS

Side effects are avoided by communicating data to and from a procedure or function through parameters. Each formal parameter listed in the heading corresponds, by position, to an actual parameter in the statement that invokes the subprogram. The following kinds of parameters can be passed:

- Value parameters
- Variable (VAR) parameters
- Procedure parameters
- Function parameters
- Conformant arrays

9.3.1 VALUE AND VAR PARAMETERS

Value and VAR parameters represent two methods of passing variable values to a procedure or function.

With a value parameter, the value of the corresponding actual parameter is copied to a second memory location when the procedure or function is invoked. The value of the value parameter is not copied back to the original memory location when the procedure or function completes. Thus, even if the parameter is assigned a new value during the execution of the procedure or function, the actual parameter remains unchanged.

An assignment to a VAR parameter during the execution of a procedure or function does change the value of the corresponding actual parameter. When a procedure or function is invoked, the address of the actual parameter is placed in the memory location reserved for a VAR parameter. Thus if a value is assigned to the VAR parameter during the execution of the procedure or function, the actual parameter itself is changed.

Since value parameters are copied for use in a procedure or function, passing large structures such as arrays or records as VAR parameters saves memory space and execution time.

Components of a packed array, components of a packed record, selectors of record variants, and CACHE variables cannot be passed as VAR parameters. When a file is passed, however, it must be passed as a VAR parameter.

The reserved word VAR precedes the variable parameters in the formal parameter list to distinguish variable parameters from value parameters. Value and VAR parameter specifications take the following forms:

```
value-parm-spec = id-list ":" type-id .

var-parm-spec = "var" id-list ":" type-id .
```

Example:

```
PROCEDURE x (i : INTEGER; VAR n, m : INTEGER);
```

Three formal parameters are declared for procedure x in the preceding example. The variable i is a value parameter, while n and m are VAR parameters.

The only restriction on the order in which the parameters appear is that the actual parameters must correspond to the positions of the formal parameters. Value and VAR parameters, as well as parameters of different types, can be interspersed as in the following example:

```
PROCEDURE y (i : INTEGER; VAR a : REAL; b,c : REAL; VAR j : INTEGER);
```

The statement that invokes a procedure or function must contain the same number of parameters listed in the procedure or function declaration, and those parameters must be of the same corresponding types. The actual parameter list in the calling statement takes the following form:

```
actual-parm-list = "(" actual-parm
                   { "," actual-parm } ")" .

actual-parm =      expression |
                   var-access |
                   procedure-id |
                   function-id .
```

An actual parameter being passed as a value parameter can be an expression, as the syntax demonstrates. An expression cannot be passed, however, as a VAR parameter. The following statement invokes procedure x of the preceding example, passing three integer parameters:

```
x (maxsize - 1, current, previous);
```

An actual parameter being passed as a value parameter must be assignment compatible with the formal parameter that corresponds to it. (Section 8, Assignment Statement and Program Control Statements, describes assignment compatibility.) An actual parameter being passed as a VAR parameter must be of the same type as the corresponding formal parameter.

NOTE

The CACHE† variable cannot be passed as a VAR parameter. CACHE variables that are larger than 1 word are copied to Common Memory when they are passed by value.

9.3.2 PROCEDURE AND FUNCTION PARAMETERS

Procedure and function names can be passed as parameters to permit procedures and functions to invoke each other without multiple definitions. In the following example, for instance, procedure processtoken calls one of three functions, depending on the contents of the array token. Which function it calls is decided in the main program and communicated to the procedure through a function parameter.

```
PROGRAM example (INPUT, OUTPUT);
TYPE tokenarray = ARRAY [1..10] OF CHAR;
VAR token : tokenarray;
  .
  .
  .
```

† Available with CRAY-2 Computer Systems only

```
FUNCTION wordcheck : BOOLEAN;
  BEGIN
  .
  .
  .
  END; (* wordcheck *)

FUNCTION numbercheck : BOOLEAN;
  BEGIN
  .
  .
  .
  END; (* numbercheck *)

FUNCTION delimcheck : BOOLEAN;
  BEGIN
  .
  .
  .
  END; (* delimcheck *)

PROCEDURE processtoken (VAR token : tokenarray; FUNCTION valid :
BOOLEAN);
  BEGIN
  .
  .
  .
  IF valid THEN ...
  .
  .
  .
  END; (* processtoken *)

BEGIN (* main program *)
  .
  .
  .
  IF token[1] IN identifiers THEN
    processtoken (token, wordcheck)
  ELSE IF token[1] IN number THEN
    processtoken (token, numbercheck)
  ELSE IF token[1] IN delimiter THEN
    processtoken (token, delimcheck)
  ELSE ...
  .
  .
  .
END. (* of program *)
```

Predefined procedures and functions cannot be passed as parameters.

9.3.3 CONFORMANT ARRAY PARAMETERS

Conformant array parameters permit a procedure or function to operate on arrays of different sizes. The declaration of a conformant array parameter in a procedure or function heading is called a *conformant array schema*, which is defined as follows:

```
conformant-array-schema = packed-conformant-array-schema |
                          unpacked-conformant-array-schema .

unpacked-conformant-array-schema = "array" "[" index-type-spec
                                     { ";" index-type-spec }  "]"
                                     "of" ( type-id |
                                     conformant-array-schema ) .

packed-conformant-array-schema = "packed" "array"
                                 "[" index-type-spec "]"
                                 "of" type-id .

index-type-spec = id ".." id ":" ordinal-type-id .
```

The identifiers that specify the lower and upper bounds of the array receive values at run time from the array that appears as an actual parameter in the function or procedure call.

Example:

```
PROGRAM conformant (OUTPUT);
   TYPE indextype = 0..10;
   VAR array1 : ARRAY [1..5] OF INTEGER;
       array2 : ARRAY [2..7] OF INTEGER;

PROCEDURE proc (x : ARRAY [low..high: indextype] OF INTEGER);
   BEGIN
      WRITELN (OUTPUT, low, high)
   END;

BEGIN
   proc (array1);
   proc (array2)
END.
```

In this example, the identifiers low and high receive values of 1 and 5, respectively, when proc is invoked the first time. On the second invocation, low and high become 2 and 7, respectively. The identifiers low and high are not proper variables and cannot be assigned values or be used to initialize constants.

An actual array must be *conformable* to the conformant array schema to be successfully passed as a parameter. An array and a schema are conformable if they meet the following criteria:

- The index types of the actual array and the schema must be compatible. (Section 8, Assignment Statement and Program Control Statements, defines type compatibility.)

- The smallest and largest index values of the actual array must lie within the range specified by the low and high bounds of the schema.

- The component types of the actual array and the schema must either be the same or be conformable. (This allows multidimensional conformant arrays, which are, strictly speaking, arrays of arrays.)

- The actual array and the schema must either be both packed or both unpacked. (In the case of a multidimensional array, only the innermost dimension can be packed.)

A conformant array formal parameter can also be passed as an actual parameter, as the following example shows:

```
PROGRAM test1 (OUTPUT);

   TYPE xx = 1..10;
   VAR q : ARRAY [1..7] OF INTEGER;

PROCEDURE p1 (VAR x : ARRAY [low1..high1: xx] OF INTEGER);
   BEGIN
      :
      :
   END;

PROCEDURE p2 (VAR y : ARRAY[low2..high2: xx] OF INTEGER);
   BEGIN
      p1(y)
   END;

BEGIN
   .
   .
   .
END.
```

When a conformant array formal parameter is passed as an actual parameter, that array cannot be passed by value. In this example, for instance, the conformant array schema x is preceded by the VAR reserved word.

The following procedure heading contains a conformant array schema that illustrates a frequent source of error in using conformant arrays.

```
PROCEDURE sample (x, y : ARRAY [lbd..hbd: indextype] OF INTEGER);
```

The two actual parameters passing values to x and y in this example schema must have exactly the same type.

CACHE† variables cannot be passed as either VAR or VALUE conformant array parameters.

9.4 PROCEDURE AND FUNCTION DIRECTIVES

Directives supply the compiler with information on the location or the characteristics of a procedure or function. The syntax for a valid procedure or function directive is as follows:

```
directive = [ ( "forward" [ ";" "exported" [ "(" external-name ")" ]
              ] ) |
              ( "exported" [ "(" external-name ")" ] [ ";" "forward" ]
              ) |
              "fortran" |
              "external" |
              ( "imported" [ "(" external-name ")" ] ) ] .

directive-alt = [ "exported" [ "(" external-name ")" ] ] .
```

9.4.1 FORWARD DIRECTIVE

The FORWARD directive is useful when a procedure or function calls another procedure or function that is defined in the same scope. Normally, a procedure or function must be declared before the statement that invokes it, but FORWARD allows a procedure or function to invoke another that is declared after the calling subprogram.

The complete heading of the subprogram to be invoked, including the parameter descriptions, is followed by the FORWARD directive and placed before the subprogram invoking it. Parameters for the forward-referenced subprogram are not repeated in the actual declaration heading.

† Available with CRAY-2 Computer Systems only

Example:

```
PROCEDURE callee (VAR x , y : INTEGER); FORWARD;

PROCEDURE caller (a, b : REAL);
BEGIN
  .
  .
  .
callee (i, k);
  .
  .
  .
END; (* of caller *)

PROCEDURE callee;
BEGIN
  .
  .
  .
END; (* of callee *)
```

The presence of the forward reference in this example enables procedure caller to invoke procedure callee. The parameters are not repeated in the declaration heading of callee.

9.4.2 EXTERNAL DIRECTIVE

The EXTERNAL directive enables a Pascal program to access routines outside of the program. EXTERNAL can be used for any routine written in Pascal or for a routine written in any other language if a compatible calling sequence is used. (See the Macros and Opdefs Reference Manual for a description of the calling sequence.) The heading of the procedure or function followed by the EXTERNAL directive replaces the declaration of the subprogram.

The EXTERNAL directive is a CRI extension to the ISO Level 1 Pascal standard.

In the following example, P$LOGMSG is the external name used by the loader.

```
PROGRAM sample (INPUT, OUTPUT);
TYPE lyne = PACKED ARRAY [1..80] OF CHAR;
.
.
.
PROCEDURE P$LOGMSG (VAR s1: lyne); EXTERNAL;
BEGIN (* main program *)
  .
  .
  .
  P$LOGMSG (param);
  .
  .
  .
END. (* of main program *)
```

Predefined procedures and functions (described in appendix B, Predefined Functions and Procedures) need not be referenced before they are used in a Pascal program.

9.4.3 FORTRAN DIRECTIVE

FORTRAN routines can be invoked from a Pascal program by using the FORTRAN directive. A heading with a FORTRAN directive is the same as a heading with an EXTERNAL directive, except that the word FORTRAN is used.

The FORTRAN directive is a CRI extension to the ISO Level 1 Pascal standard.

Example:

```
FUNCTION f (i : INTEGER; VAR r : REAL) : REAL; FORTRAN;
```

As the example indicates, value parameters can be passed to FORTRAN subprograms.

Problems may arise when calling FORTRAN routines because of incompatibilities between Pascal and FORTRAN. The following are potential difficulties:

- Arrays are stored differently in FORTRAN than in Pascal. Passing multidimensional arrays such as the following may cause confusion:

```
VAR a : ARRAY [0..10, 1..20] OF INTEGER;
```

This declaration must be interpreted by the FORTRAN routine as follows:

```
INTEGER A (20, 0:10)
```

An array accessed as A[I, J, K] in Pascal is referenced as A(K, J, I) in a FORTRAN routine. Either the Pascal program or the FORTRAN routine must transpose such arrays.

- The FORTRAN type COMPLEX can be declared as a record with two fields of type REAL. The result of a FORTRAN function, however, cannot be COMPLEX, because Pascal does not allow function results of type RECORD. Thus, a Pascal program cannot call FORTRAN functions of type COMPLEX but can pass VAR parameters of type COMPLEX if they are first declared as records.

- Pascal procedures and functions cannot be passed as parameters in a FORTRAN declaration. But external FORTRAN subroutines and functions can be declared as procedure and function parameters.

- The Pascal compiler does not test for inappropriate parameter types when calling FORTRAN routines. INTEGER, REAL, COMPLEX, ARRAY, SUBROUTINE, and FUNCTION are types generally suitable for a FORTRAN routine called from a Pascal program. (Other types can be passed if you know how storage is allocated by both the Pascal compiler and the FORTRAN compiler and can perform any necessary conversions.)

- Pascal conformant arrays cannot be passed to FORTRAN routines.

- FORTRAN variable-length characters arguments cannot be passed from Pascal procedures.

9.4.4 IMPORTED AND EXPORTED DIRECTIVES

The IMPORTED and EXPORTED directives enable a Pascal program to share procedures and functions defined in separate compile units. (Section 12, Modules, describes module compile units, including examples of IMPORTED and EXPORTED directives.)

The IMPORTED and EXPORTED directives are CRI extensions to the ISO Level 1 Pascal standard.

The EXPORTED directive appears in the heading of the procedure or function to be accessed. Exported routines are normally library routines. The IMPORTED directive appears in the program or module that will use the routine. Only the procedure or function heading and the IMPORTED directive can appear in a program importing a routine. The routine is executed when the procedure or function name is encountered in the executable statements section of the program.

The syntax for the IMPORTED directive is as follows:

```
"imported" [ "(" external-name ")" ] .
```

The EXPORTED directive identifies a procedure or function to be used outside the compile unit in which it is defined. A procedure or function can only be EXPORTED from the outermost nesting level of a module or program. Inner level procedures and functions can be called by an exported routine but cannot themselves be exported.

The syntax for the EXPORTED directive is as follows:

```
"exported" [ "(" external-name ")" ] .
```

The external name in an IMPORTED directive is, if specified, the name by which the loader knows a routine. The name must be within the loader limitation of 8 characters for CRAY X-MP and CRAY-1 Computer Systems and 32 characters for CRAY-2 Computer Systems. If no external name was assigned to an IMPORTED or EXPORTED procedure or function, the name of the routine is used with truncation where the name exceeds the external name limitations.

Example:

In program a:

```
PROCEDURE cat (i : INTEGER); IMPORTED (fred);
PROCEDURE dog (j : INTEGER); IMPORTED (joe);
.
.
.
```

In module b:

```
PROCEDURE turkey (k: INTEGER); EXPORTED (fred);
.
.
.
PROCEDURE joe (n : INTEGER); EXPORTED;
.
.
.
```

In this example, the external name fred matches in procedures cat and turkey. The loader uses the name fred to execute the procedure, while the Pascal program continues to know the routine as cat. When cat is encountered in the executable statements section of the program, the loader begins execution on the routine known to it as fred. The exported procedure joe has no external name and must therefore be matched by joe in the IMPORTED directive in the program. If the name joe were josephine instead, it would be truncated to the 8 characters josephin for use by the loader on a CRAY-1 or CRAY X-MP Computer System.

If the external name is missing from the IMPORTED directive, the procedure or function name is used. The procedure or function name must then match the name by which the loader knows the exported routine. Again, if the name is longer than 8 characters on a CRAY-1 or CRAY X-MP, it is truncated to 8 characters for use by the loader.

The capability of renaming an imported routine within the scope of the program makes using the same routine under two different names possible. In the following example, routine ISRCHEQ in $SCILIB is referenced in two procedures:

```
PROCEDURE vsearch_int ( VAR limit,
                        firstelement,
                        step,
                        target : INTEGER); IMPORTED(ISRCHEQ);

PROCEDURE vsearch_real ( VAR limit : INTEGER;
                         VAR firstelement : REAL;
                         VAR step : INTEGER;
                         VAR target : REAL); IMPORTED(ISRCHEQ);
```

By declaring ISRCHEQ in both ways, both real and integer vectors can be searched without argument type conflicts.

The level to which procedures and functions are nested within a program is not affected by imported routines. For instance, if a procedure at a nesting level of 10 in a program imports a module with three levels of procedures, the nesting level of the program remains at 10 despite the imported module.

The FORWARD directive can be used with the EXPORTED directive by inserting the following either before or after the EXPORTED directive:

```
FORWARD;
```

When using procedures or functions with both EXPORTED and FORWARD directives, the rules for FORWARD procedures and functions must be followed when establishing the actual block containing the code for the routine. (The FORWARD directive is described earlier in this section.)

9.5 RECURSIVE PROCEDURES AND FUNCTIONS

A Pascal procedure or function can invoke itself. Such an invocation is termed a *recursive* call.

Example:

```
FUNCTION factorial (current_value : INTEGER): INTEGER;
BEGIN
  IF current_value = 0 THEN
     factorial := 1
  ELSE
     factorial := factorial (current_value - 1) * current_value
  END
END;
```

This is an example of direct recursion; the function factorial contains its own invocation. The following example is an instance of indirect recursion; procedure recur1 invokes procedure process_partial, which contains a call to recur1. A FORWARD directive (described earlier in this section) is required for recur1.

```
PROCEDURE recur1 (pram1 : REAL); FORWARD;

PROCEDURE process_partial (pram2 : REAL);
  .
  .
  .
  recur1(running_total);
END;

PROCEDURE recur1;
  .
  .
  .
   IF ... THEN process_partial(partial_total);
  .
  .
  .
END;
```

Each recursive call must be contained within a conditional structure that avoids infinite recursion by preventing the execution of the call at some point.

Information on recursive techniques is available in Pascal textbooks such as Peter Grogono's, *Programming in Pascal*.

9.6 PREDEFINED PROCEDURES AND FUNCTIONS

Predefined Pascal procedures and functions need not be declared in a Pascal program before they are invoked. Appendix B, Predefined Functions and Procedures, describes the predefined procedures and functions.

You can redefine any of these procedures and functions, using the same name. The routine you supply is executed when invoked instead of the predefined routine.

10. INPUT AND OUTPUT

Input and output statements move data between local files or external devices, such as disks, and data structures within a Pascal program. These statements read data from and write data to file types, which must be declared (see section 5, Data Types) unless they are predefined types.

Pascal provides two predefined text files, INPUT and OUTPUT, that must be named as parameters in the program heading (see section 4, Program Organization) before they are used.

Data is passed between files and internal data structures through a buffer variable (see section 5, Data Types). The predefined I/O procedures GET and PUT read or write a single component into or from the buffer variable. Higher level I/O procedures (READ, READLN, WRITE, and WRITELN) move data directly between a file and a variable.

Pascal does not directly support tape I/O. However, FORTRAN-callable library routines to perform tape I/O can be invoked using the FORTRAN directive. (Section 9, Procedures and Functions, describes the FORTRAN directive.)

10.1 PREDEFINED FILES INPUT AND OUTPUT

INPUT and OUTPUT are predefined text files. (Text files are described in section 5, Data Types.)

The presence of INPUT as a parameter in the program heading indicates that the $IN (COS) or stdin (UNICOS) dataset includes a text file of data. OUTPUT specifies that a file of output data will be included in the $OUT (COS) or stdout (UNICOS) datasets.

As text files, INPUT and OUTPUT must contain constants of type CHAR and end-of-line indicators. Unlike other text files, INPUT and OUTPUT do not require the use of predefined procedures RESET and REWRITE, respectively. (RESET and REWRITE are described later in this section with the GET and PUT procedures.)

INPUT is the default file in the READ and READLN statements as well as in the EOF and EOLN functions. OUTPUT is the default file in the WRITE and WRITELN statements. The READ, READLN, WRITE, and WRITELN statements are described later in this section.

10.2 BUFFER VARIABLE

A file is accessed sequentially through a buffer variable, into which file elements are read or written one at a time.

The position of the buffer variable in the file may be thought of as a window (see section 5, Data Types). The window advances through the file element by element to allow data to be read from or written to each file position. When an element of data is read from the window position in a file, it is read into the buffer variable; from there it can be assigned to an element in an array, for instance. For a write operation, the data is first assigned to the buffer variable and then written to the window position at the end of the file.

A buffer variable of the same type as the file is created implicitly for each file declared. The buffer variable is referenced using the following form:

```
file-name "^" .
```

The following statement, for example, assigns the value of the variable x to the buffer variable for the file outfile:

```
outfile^ := x;
```

10.3 GET AND PUT

The GET and PUT procedures combine the operations of advancing the window of a text file and reading from or writing to the buffer variable.

GET moves the window to the next element of the file specified as the parameter and assigns the value of that element to the buffer variable. If the next element is the end-of-file, the buffer variable is undefined and the end-of-file condition is set to TRUE.

PUT writes the contents of the buffer variable to the window position at the end of the file specified as the parameter. The end-of-file condition must be true before PUT is executed; if the end-of-file condition is not true, a run-time error occurs.

The GET and PUT procedures are invoked with statements of the following form:

```
"get" "(" file-var ")" .
"put" "(" file-var ")" .
```

Before using GET or PUT on any file except INPUT or OUTPUT, the file is prepared for reading or writing by one of the predefined procedures for that purpose, RESET or REWRITE. Using RESET on OUTPUT or REWRITE on INPUT causes an error.

RESET sets the window at the beginning of the specified file before reading from that file. The end-of-file condition is set to FALSE and the value of the first element in the file is assigned to the buffer variable, unless the file is empty.

REWRITE deletes the contents of the specified file before writing to that file. The end-of-file condition is set to true.

The following example copies the contents of a file called source into a file called target:

```
RESET (source);
REWRITE (target);
WHILE NOT EOF(source) DO
BEGIN
  target^ := source^;
  PUT(target);
  GET(source)
END; (* WHILE loop *)
```

10.4 READ AND WRITE

READ and WRITE are predefined procedures that go a step further than the GET and PUT procedures by eliminating any explicit reference to the buffer variable. READ moves data directly between an external file and variables internal to a Pascal program. WRITE moves data in the other direction, from variables to a file.

A READ(source,x) statement is equivalent to the following statements:

```
x := source^;
GET(source)
```

Similarly, the following sequence represents a WRITE(target,x) statement:

```
target^ := x;
PUT(target)
```

A file name must be specified in a READ or WRITE procedure call only if the predefined files INPUT and OUTPUT are not used. INPUT is the default file for READ and OUTPUT for WRITE.

```
"read" "(" [ file-var "," ] var-id { "," var-id } ")" .

"write" "(" [ file-var "," ] element { "," element } ")" .
```

Input and output items (var-id on a READ and element on a WRITE) must match the data type of the file unless the file is type TEXT. Data of any type can be read from or written to a text file; Pascal converts the data to or from its text representation.

As with GET and PUT, READ and WRITE must employ the RESET or REWRITE procedure before beginning I/O with any file other than INPUT or OUTPUT.

Example:

```
RESET (source);
REWRITE (target);
READ(source,ch);
WRITE(target,ch);
```

READ variables are associated with data according to position; that is, the first data element encountered is read into the first variable, the second element into the second variable, and so on. With the exception of elements of type CHAR, data elements being read must be separated by at least 1 non-numeric character (such as a blank). The exception is possible because elements of type CHAR are known to be 1 character in length.

The type of variable appearing in a READ or WRITE statement must agree with the type of the data being read or written. If the types do not agree and run-time checking is enabled, the program aborts with an error message. The following line of data in the INPUT file would be read correctly by a READ(i, x, ch, b) statement if i is declared as type INTEGER, x as type REAL, ch as type CHAR, and b as type BOOLEAN:

```
36  3.14  A  TRUE
```

Variables i, x, ch, and b are assigned values of 36, 3.14, A, and TRUE, respectively.

If the READ statement contains more variables than the line has input items, the READ procedure skips to the next line to find the extra values. If, for example, the statement READ(x, y, z) is used on the following input file, x has a value of 6, y of 9, and z of 44:

```
6    9
44   13
```

String variables can also be specified with the READ statement. In such instances, as many characters as necessary are read to fully define the variable. If EOLN is encountered before the string has been fully defined, the remaining elements in the variable are filled with blanks.

Data to be written with a WRITE statement can be represented in either of the following forms:

- As a literal. An element delimited by apostrophes is written to the output file exactly as specified. If an apostrophe is part of the literal, it is represented by two apostrophes. The following statement writes the characters abc followed by a blank:

```
WRITE ('abc ');
```

- As an expression, the most simple form of which is a variable. The following statement writes two valid values, assuming the variables involved are properly declared:

```
WRITE (ch, x * 2);
```

10.5 READLN AND WRITELN

The READLN and WRITELN procedures are variants of the READ and WRITE procedures for use with files containing end-of-line designators. The syntax is the same for READ and READLN and for WRITE and WRITELN.

Cray Pascal supports COS blocked format (single-file datasets) when executing under COS. Blocked datasets are described in the COS Version 1 Reference Manual. Unblocked (multiple-file) datasets are supported only through FORTRAN library routines described in the System Library Reference Manual. When using these FORTRAN routines on unblocked datasets, you are responsible for buffer management. (Section 9, Procedures and Functions, describes how to call FORTRAN library routines.)

Cray Pascal supports UNICOS unblocked files when executing under UNICOS. Unblocked files are described in the CRAY-2 UNICOS Libraries, Macros and Opdefs Reference Manual, publication SR-2013. If blocked files are specified, you are responsible for all buffer management. The UNICOS File Formats and Special Files Reference Manual describes the format for blocked files.

READLN reads values for the variables specified as parameters and moves the window to the next line when complete. READ does not move the window to the next line after a completed operation unless it has just read the last element in a line. Assume, for example, the following lines of integers on an input file:

```
1    3    5    7    9    11   13   15
17   19   21   23   25   27   29   31
```

The following statements will read part of the preceding data, assuming the variables have been declared as type INTEGER:

```
READLN (int1, int2, int3);
READLN (int4, int5, int6);
```

These statements assign the variables the following values:

```
int1      1
int2      3
int3      5
int4      17
int5      19
int6      21
```

The same variables contain different values if read by equivalent READ statements such as the following:

```
READ (int1, int2, int3);
READ (int4, int5, int6);
```

These READ statements do not advance the window to the next line. The values of the variables are as follows:

```
int1      1
int2      3
int3      5
int4      7
int5      9
int6      11
```

The following example uses both READ and READLN to read a text file and process its characters:

```
RESET (infile);
WHILE NOT EOF(infile) DO
  BEGIN
    WHILE NOT EOLN (infile) DO
      BEGIN
        READ (infile, ch);
        process (ch)
      END;
    READLN (infile)
  END;
```

WRITELN writes an end-of-line designator to the output file, terminating the present line.

Example:

```
position := 1;
REWRITE (outfile);
FOR lines := 1 TO maxlines DO
  BEGIN
    FOR position := 1 TO 80 DO
      WRITE (outfile, line[position]);
    WRITELN (outfile)
  END;
```

10.6 FORMATTING OUTPUT

Formatting specifications are available with the WRITE and WRITELN statements that permit the appearance of Pascal output to be manipulated. The form of an output element for a real number is as follows:

```
element = 'constant' | expression
          [ ":" field-width ":" [ decimal-places ] ] .
```

The following form is used for integers and other scalar data types:

```
element = 'constant' | expression [ ":" field-width ] .
```

The field width is a decimal number specifying the minimum number of spaces to be filled by the output. If fewer spaces are required than the number specified, the output is padded with leading blanks.

When a character string or packed array of characters requires more output spaces than the number specified, only the number of characters that fit in the specified space are written. If, however, an output item other than a character string or packed array of characters is larger than the specified output space, the entire value is printed.

The decimal-places specification determines how many places to the right of the decimal point are represented for a real number. Unless the decimal-places specification is included, a real number is represented by a coefficient and a scale number (such as 5E-8).

Example:

```
WRITELN ('  The value of ''x'' is', x:6:2, '.');
```

Assuming the value of x is 8.9, the output line for the preceding example appears as follows:

```
  1 . . . 5 . . . . 10 . . . .
| ␣ ␣ T h e ␣ v a l u e ␣ o f /
                                  15 . . . . 20 . . . . 25 . .
                                / ␣ ' x ' i s ␣ ␣ 8 . 9 0 . |
```

If the field-width is not specified, output elements are assigned default field width values as follows:

Type	Output Width
Boolean	10
Character	1
Integer	10
Real	22, in the following format (position 1 is the leftmost position):

Bit	Description
1	Minus sign or blank
2	One digit
3	Decimal point
4-16	13 digits
17	E
18	Plus or minus sign
19-22	Four digits

Example:

```
0.1932792849915E-0006
```

String or packed array of characters
Length of string or number of elements in array

The predefined procedure PAGE (see appendix B, Predefined Functions and Procedures) writes an end-of-line and puts the carriage control character that forces a page break into the output file.

10.7 CONNECT

Files appearing as parameters in the program heading are bound to COS local datasets or UNICOS files of the same name. Files not appearing as parameters in the program heading are bound to unique temporary (scratch) datasets. The CONNECT procedure overrides these bindings and permits you to specify a file name other than the local dataset name.

CONNECT is a CRI extension to the ISO Level 1 Pascal standard.

(A COS dataset is made local to a job when it is referenced in a job control statement such as ACCESS or ACQUIRE. See the COS Version 1 Reference Manual for more information on local datasets and the ACCESS and ACQUIRE control statements.)

The form of a call to the CONNECT procedure is as follows:

```
"connect" "(" file-var "," "'" local-dataset-name "'" ")" .
```

For COS, the local dataset name is passed to an 8-character packed array. Thus, the name must appear left-justified and padded with blanks in the procedure call. The file can contain data of any valid type.

For UNICOS, the file name can be of any length but must be left-justified if blanks are used. File names that are not full path names default to the local directory. File names must be terminated with a least one blank character, which is not considered as part of the file name.

If you specify file names $IN, $OUT, and $LOG, you connect to the UNICOS standard input, output, and error files, respectively.

COS example:

```
CONNECT (infile,'FRED    ');
```

The Pascal file variable infile, which must first be declared in a VAR statement, represents the local dataset FRED after the above statement is executed. Data can subsequently be read from or written to FRED. Before I/O can begin, however, the file must be prepared for reading or writing by one of the predefined procedures, either RESET or REWRITE. In the following example, for instance, data cannot be read from the local dataset GEORGE until a second RESET statement appears. Any statements that read from infile following the second CONNECT statement would continue to read from the local dataset FRED until a second RESET statement appears.

```
CONNECT(infile,'FRED    ');
RESET(infile);
.
.
.
CONNECT(infile,'GEORGE  ');
```

UNICOS examples:

```
CONNECT(france,'gaul ');
CONNECT(england, 'dearoldalbion ');
```

11. DYNAMIC ALLOCATION

Pascal permits the allocation of memory for a variable either at the time a program is being compiled or dynamically during the execution of the program.

Variables declared in a VAR statement are allocated memory when the program is compiled. Dynamically allocated variables are not allocated memory, however, until the executing program requests them through the predefined procedure NEW. The variables become active following the execution of the NEW procedure, and they remain active until the executing program deallocates their space by calling the predefined procedure DISPOSE. The deallocated space is then available for reuse by subsequently allocated variables.

You do not supply a name by which to reference a dynamically allocated variable. The variable is referenced indirectly through a pointer, which is assigned the memory address of a dynamically allocated variable when that variable is allocated. Before a dynamically allocated variable is defined or after it is deallocated, the pointer that references it has an undefined value. The special value NIL, which corresponds to no memory address, can also be assigned to a pointer. Section 5, Data Types, describes the declaration of a pointer variable.

Procedures NEW and DISPOSE allocate and deallocate variables. (*p* represents a pointer variable.) Function SIZEOF gives the size of a dynamic variable. A description follows:

Function or Procedure	Description
NEW(*p*) (procedure)	Defines a dynamically allocated variable pointed to by *p*. The dynamically allocated variable is of the type to which *p* is bound.
NEW(*p*, *tag-field-value*$_1$, *tag-field-value*$_2$, ... *tag-field-value*$_n$) (procedure)	Defines a dynamic record with variants pointed to by *p*. Space allocated for the variant fields depends on the tag field values listed. The tag field values must be listed in the order declared, and the list must be contiguous. The first tag field value must correspond to the first tag field declared, but the last tag field value in the list need not be the last tag field declared.

Function or Procedure	Description
DISPOSE(*p*) (procedure)	Deallocates the dynamically allocated variable pointed to by *p*. *p* retains the same value, but *p*^ becomes undefined.
SIZEOF(*p, tag-field-value*$_1$, *tag-field-value*$_2$, ... *tag-field-value*$_n$) (function)	Returns the size (in Cray words) of a dynamic variable: that is, the integer number of words that a call to NEW allocates when called with the same parameter list. SIZEOF takes a variable number of parameters and works exactly like NEW used as a function. The value of pointer *p* is returned unaltered.

The SIZEOF function is a CRI extension to the ISO Level 1 Pascal standard.

Example:

```
TYPE pntr = ^personnel;
     personnel = RECORD
                 link: pntr;
                 .
                 .
                 .
                 END;
VAR p1, p2, first: pntr;
```

This example declares a pointer type named pntr and three pointer variables named p1, p2, and first that are bound to the record type named personnel. The record itself contains a pointer variable named link. The first record is dynamically allocated in the following manner:

```
NEW(first);
first^.link := NIL;
```

The first statement dynamically allocates one record of type personnel and sets the pointer first to point to it. The second statement accesses the link field of that record through the pointer and assigns the value NIL to it. A second record can be dynamically allocated and linked to the first as follows:

```
p2 := first;
NEW(p1);
p1^.link := p2^.link;
p2^.link := p1;
```

The list of records is linked through the link fields. Records are added in the middle of such a list by the following steps:

1. Determine the record after which the new record is to be added, traversing the list by means of the link field pointers.

2. Set the pointer p2 to point to that record.

3. Use the following statements to dynamically allocate and insert the new record:

```
NEW(p1);
p1^.link := p2^.link;
p2^.link := p1;
```

To delete a record, pointer p2 is again set to point to the record that precedes the one to be deleted. The following statements delete the record while maintaining the list in order:

```
p2^.link := p1^.link;
DISPOSE(p1);
```

The second form of the NEW statement operates in the same way as the first form, but it also permits tag field values to be specified for variant records.

Example:

```
TYPE pntr = ^personnel;
     personnel = RECORD
                   link: pntr;
                   .
                   .
                   .
                 CASE pension: vested OF
                   unvested;
                   partvest : (pmoney, percentage: REAL);
                   fullvest : (fmoney: REAL;
                               yearvested: INTEGER);
                               CASE retired: BOOLEAN OF
                                 FALSE ( );
                                 TRUE (pension_monthly_amt: REAL);
                 END
VAR p1, p2, first: pntr;
```

A record for a fully vested, currently active employee is allocated as follows:

```
NEW(p1, fullvest, FALSE);
```

A record for a fully vested, retired employee is reallocated as follows:

```
NEW(p1, fullvest, TRUE);
```

A record including the fields fmoney, yearvested, and pension_montly_amt is allocated by the above statement. The variant for a record allocated by this form of the NEW statement cannot be changed later in the program. The above record, for instance, cannot be changed later to contain the fields designated by the tag field value partvest.

If a variant record is dynamically allocated by the first form of the NEW statement (specifying no tag field values), enough space is allocated to accommodate the largest possible requirements.

12. MODULES

The Pascal compiler accepts two types of compilation units: program and module. A module is a stand-alone routine, such as a library routine, that can be accessed and executed by other modules and programs. A module allows the separate compilation of encapsulated code and data.

The module compile unit is a CRI extension to the ISO Level 1 Pascal standard.

Modules are normally defined as library datasets. For information on generating and maintaining library datasets under COS, see the description of BUILD in the COS Version 1 Reference Manual. The ar command creates libraries under UNICOS. (When using the UNICOS link editor ld, specify the library containing modules with the -L option rather than the -l option.)

In structure, a module is similar to a program, except that a module has no program parameters and no main program block delimited by the reserved words BEGIN and END. A module contains a series of internal procedures and functions and also accepts external routines referenced from programs or other modules.

The syntax of a module is as follows:

```
module = "module" id ";" module-block "." .

module-block =      {
                     constant-def-part
                     type-def-part
                     common-dcl-part
                     imported-dcl-part
                     exported-dcl-part
                     static-dcl-part
                     var-dcl-part
                    }
                    [ value-def-part ]
                    { procedure-and-function-dcl-part } .
```

A module cannot be directly referenced by a caller. Rather, the procedures and functions at the outermost level in a module can be exported and thereby referenced (as imported) by other modules and programs. (See section 9, Procedures and Functions, for imported and exported routines.) Procedures and functions that are not at the outermost level can be referenced by exported procedures and functions but cannot be exported themselves.

Data is communicated between modules and programs strictly through parameters. A module called from a program cannot make use of global variables declared in the program unless they are passed as parameters. The constants and types defined at the module level allow the user to pass structured parameters, such as records.

NOTE

No type checking is performed between modules. Hence, you must ensure type compatibility between modules and programs by using some method of common TYPE declarations.

Variables declared at the module level are global within the module. This allows the normal scope rules to apply within the module but hides module variables from routines outside the module. Since module variables are allocated in static fashion (not on the run-time stack), the variables retain their values from call to call.

Modules do not support the predefined text files INPUT and OUTPUT. You must pass these files in as parameters. In the following example, a hidden stack is used to hold tokens for evaluation. If more than one stack is needed, the stacks can be declared in the program and the code modified to pass the desired stack to the STK$ routines.

```
PROGRAM pushpop(INPUT,OUTPUT);
  TYPE token = PACKED ARRAY[1..8] OF CHAR;
  VAR t,t1,r : token;

  PROCEDURE push( t : token );IMPORTED(stk$push);
  PROCEDURE pop ( VAR t : token );IMPORTED(stk$pop);
  PROCEDURE stackinit;IMPORTED(stk$init);
  FUNCTION  empty: BOOLEAN;IMPORTED(stk$mt);
  FUNCTION  full : BOOLEAN;IMPORTED(stk$full);
  PROCEDURE gettok(VAR t : token;
                   VAR fromfile:TEXT); IMPORTED(token);
  PROCEDURE crunch(VAR t1,t2,t3 : token); IMPORTED(doittoit);
```

```
PROCEDURE error(i : INTEGER);IMPORTED;
BEGIN (*Main program code *)
  stackinit; (*Initialize the stack*)
  gettok(t,INPUT);
  WHILE t <> '        ' DO
    BEGIN
      IF t = '(       '
      THEN push(t)
      ELSE IF t = ')       '
           THEN BEGIN
                  pop(t1);
                  crunch(t,t1,r);
                  push(r)
                END
           ELSE error(1); (*Token ignored*)
      gettok(t,INPUT);
    END;
  IF NOT empty
  THEN error(2)  (* Token stack not empty at end*)
  ELSE WRITELN(' The answer is : ',r);
END.
```

The procedures used in this program could be coded in a module such as the following. This module declares some of the routines mentioned in the program and references others through the IMPORTED directive.

```
MODULE stk;
TYPE token = PACKED ARRAY[1..8] OF CHAR;
     astack = RECORD
                stackptr : INTEGER;
                stack : ARRAY[1..1024] OF token;
              END;
VAR thestack : astack;

PROCEDURE stk$init; EXPORTED;
BEGIN
  thestack.stackptr := 0;
END;

FUNCTION stkfull:BOOLEAN; EXPORTED(stk$full);
BEGIN
  stkfull := thestack.stackptr > 1024;
END;

PROCEDURE stkempty : BOOLEAN;EXPORTED(stk$mt);
BEGIN
  stkempty := thestack.stackptr = 0;
END;

PROCEDURE error( i : INTEGER); IMPORTED;
```

```
  PROCEDURE push ( t : token ); EXPORTED(stk$push);
  BEGIN
    IF stkfull
    THEN BEGIN    (* WHOOPS! *)
           error(2);
           HALT;
         END
    ELSE WITH thestack DO
           BEGIN
             stackptr := stackptr + 1;
             stack[stackptr] := t;
           END;
  END;

  PROCEDURE pop( VAR t : token); EXPORTED(stk$pop);
  BEGIN
    IF stkempty
    THEN BEGIN  (* WHOOPS! *)
           error(3);
           HALT;
         END
    ELSE WITH thestack DO
           BEGIN
             t := stack[stackptr];
             stackptr := stackptr - 1;
           END;
END.
```

13. VECTORIZATION AND OPTIMIZATION

This section describes the following:

- Vectorization:
  - How Pascal vectorizes code
  - Constructs that prohibit vectorization
- Optimization:
  - High- and low-level
  - Enabling and disabling optimization using the Pascal invocation statement

13.1 VECTORIZATION

The Pascal compiler converts some FOR loops into sequences of vector operations. At run time, vectorized FOR loops use the vector registers and functional units of the Cray hardware to greatly reduce execution time. Vectorization is enabled by the V+ compiler option, which can appear on the Pascal invocation statement or in a compiler directive appearing between compilation units. (Vectorization is enabled by default.)

While the Pascal compiler attempts to vectorize every FOR loop, not every FOR loop can be vectorized. Some loops, if vectorized, produce results that differ from those generated by scalar code; other loops contain operations that cannot be performed in vector mode. The Pascal compiler performs a detailed analysis of each FOR loop and does not vectorize loops for which the scalar-to-vector transformation cannot be safely guaranteed.

The following constructs inhibit vectorization within a FOR loop:

- Procedure calls, including standard I/O procedures such as READ and WRITE
- Function calls other than calls to the standard mathematical functions

- Any statement other than an assignment statement, an IF statement, or a compound statement containing only assignments and IF statements
- Run-time array bounds checking (enabled by default)
- Any operand that does not fall into one of the following categories:

Category	Description
Constant	A literal constant

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
BEGIN
   FOR i := 1 TO n DO
   a [i] := 0.0;
```

In the previous example, 0.0 is a constant.

Invariant — An invariant is defined as follows:

- A variable that is not assigned within the loop

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   x : REAL;
BEGIN
   FOR i := 1 TO n DO
   a [i] := a [i] + x;
```

In the previous example, x is the invariant.

- A pointer dereferencing (an access of a variable identified by a pointer) or field access whose base variable is not assigned within the loop

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   p : ARRAY [ 1..n ] OF ^ INTEGER;
   r : ARRAY [ 1..n ] OF RECORD
     f, g : INTEGER;
BEGIN
   FOR i := 1 TO n DO
   a [i] := p^ + r.f;
```

Category	Description

In the previous example p^ and r.f are the invariants, because neither is altered within the loop.

- An array element access whose subscript expression and base variable are not assigned within the loop

```
VAR
   a, b : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
BEGIN
   FOR i := 1 TO n DO
   a [i] := b [10];
```

In the previous example, b [10] is invariant, because an element is accessed whose subscript is not assigned within the loop.

- An expression containing operations on only constants and operands not assigned within the loop

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   p : ARRAY [ 1..n ] OF ^ INTEGER;
   x : REAL;
BEGIN
   FOR i := 1 TO n DO
   a [i] := a [i] * (x + p^);
```

In the previous example, x, p^, and (x + p^) are invariant, because x is not assigned within the loop, p^ is a pointer access not assigned within the loop, and (x + p^) is an expression that contains only invariants.

Loop control — A FOR loop control variable

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
BEGIN
   FOR i := 1 TO n DO
   a [i] := 0.0;
```

In the previous example, i is the loop control variable.

Category	Description

Vector definition

A vector definition is defined as follows:

- An array subscript containing only the FOR loop control variable and constant operands (and invariant operands if the VI option is placed in a directive preceding the loop)

```
VAR
   a, b : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
BEGIN
   FOR i := 1 TO n DO
   a [i] := b [i - 10];
```

  In the previous example, b [i - 10] is a vector definition, because it contains only the FOR loop control variable (i) and a constant (10).

- A pointer dereferencing or field access whose base variable is a vector definition

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   p : ARRAY [ 1..n ] OF ^ INTEGER;
   r : ARRAY [ 1..n ] OF RECORD
     f, g : INTEGER;
BEGIN
   FOR i := 1 TO n DO
   a [i] := p [i]^ * r [i].f;
```

  In the previous example, p [i]^ and r [i].f are vector definitions, because p [i]^ is a pointer dereferencing and r [i].f is a field access whose base variable (r [i]) is a vector definition.

- An array element access whose subscript expression is invariant and whose base variable is a vector definition

Category	Description

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   x : REAL;
   p : ARRAY [ 1..n ] OF ^ INTEGER;
BEGIN
   FOR i := 1 TO n DO
   a [i] := a [i] + x + p [i]^;
```

In the previous example, a [i] and p [i]^ are vector definitions, because a [i] is a vector definition (an array element access with an invariant subscript) and p [i]^ is a pointer dereferencing whose base variable (p [i]) is a vector definition.

Vector expression

A vector expression is an expression containing operations on constant, invariant, vector definition, and vector expression operands

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   x : REAL;
   r : ARRAY [ 1..n ] OF RECORD
     f, g : INTEGER;
BEGIN
   FOR i := 1 TO n DO
   a [i] := r [i].g + x
```

In the previous example, r [i].g is a vector expression because r [i].g is a field access whose base variable is a vector definition.

Vector assignment

A vector assignment is an assignment statement whose left side is a vector definition and whose right side is a constant, an invariant, a vector definition, or a vector expression

```
VAR
   i, n : INTEGER;
   r : ARRAY [ 1..n ] OF RECORD
     f, g : INTEGER;
BEGIN
   FOR i := 1 TO n DO
   r [i].f := 1
```

Category	Description

In the previous example, r [i].f := 1 is a vector assignment, because r [i].f is a vector definition and 1 is a constant.

Vector IF

A vector IF is an IF statement whose conditional expression is a vector definition or a vector expression and whose THEN and ELSE parts contain only vector assignments

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   p : ARRAY [ 1..n ] OF ^ INTEGER;
BEGIN
   FOR i := 1 TO n DO
   IF p [i] = NIL THEN
     a [i] := 0.0
   ELSE
     a [i] := p [i]^;
```

Reduction

A reduction is an assignment statement that is the only statement in the loop and has the following form:

```
var := ([var] [op] [exp])
```

var is a simple variable

op is one of the following: +, *, -, AND, or OR

exp is a vector expression or a vector definition

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   x : REAL;
BEGIN
   FOR i := 1 TO n DO
   x := x + a [i];
```

Search

Search is an IF statement whose condition is a vector expression or vector definition containing only a GOTO statement.

Category	Description

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   x : REAL;
BEGIN
   FOR i := 1 TO n DO
   IF a [i] = x THEN
     GOTO 1;
```

- Vector dependencies may exist in a loop if vectorization does one of the following:

  - Overwrites a word of memory before using it
  - Uses a word of memory before storing a value into it

  Vector dependencies arise in one of four ways:

  1. Negative offset before a store in an incrementing loop:

     ```
     FOR i := 2 TO n DO
       a [i] := a [i-1];
     ```

  2. Positive offset after a store in an incrementing loop:

     ```
     FOR i := 1 TO n-1 DO
     BEGIN
       a [i] := 0;
       b [i] :=a [i+1]
     END;
     ```

  3. Positive offset before a store in a decrementing loop:

     ```
     FOR i := n-1 DOWNTO 1 DO
       a [i] := a [i+1];
     ```

  4. Negative offset after a store in a decrementing loop:

     ```
     FOR i := n DOWNTO 2 DO
     BEGIN
       a [i] := 0;
       b [i] :=a [i-1]
     END;
     ```

  Pascal analyzes each FOR statement for explicit or potential dependencies prior to vectorizing the loop unless the VI compiler option is used in a comment immediately preceding the loop.

Dependencies are detected by comparing every vector definition on the left side of an assignment statement to every other vector definition in the loop. A dependency exists if the following conditions exist:

1. Both vector definitions could refer to the same array

2. The offsets used in the subscript expressions could produce one of the four dependencies described previously

Pascal assumes a dependency may exist if the subscripts or base variables are ambiguous. If there is any possibility that a dependency exists, Pascal does not vectorize a FOR statement. Pascal does not vectorize the following FOR statements, because dependencies may exist within the loops:

```
FOR i := 1 TO n DO
  a [i] := a[i+b];              (* b could be less than 0 *)

FOR i := 1 TO n DO
  p^ [i] := q^ [i-1];           (* p could equal q *)

FOR i := 1 TO n DO
  a [i+n] := a [i];             (* n could be greater than 0 *)

FOR i := 1 TO n DO
  a [i] := a [b [i] ];          (* b [i] could be anything *)
```

Example:

```
VAR
   a : ARRAY [ 1..n ] OF REAL;
   i, n : INTEGER;
   x : REAL;
   p : ARRAY [1..n] OF ^ INTEGER;
   r : ARRAY [ 1..n ] OF RECORD
     f, g : INTEGER;
   END;
BEGIN
   FOR i := 1 TO n DO                    (* Line 1 *)
     BEGIN                               (* Line 2 *)
     a [i] := 100.0;                     (* Line 3 *)
     IF p [i] = NIL THEN                 (* Line 4 *)
       r[i].f := r[1].g                  (* Line 5 *)
     ELSE                                (* Line 6 *)
       r[i].f := r[i].f * p[i]^          (* Line 7 *)
     END;                                (* Line 8 *)
END;
```

In the previous example, the operands in the loop are classified as follows:

Line Number	Class	Operand
1	Loop control	i
2	Constant	100.0
3	Vector definition	a [i]
3	Vector assignment	a [i] := 100.0
4	Vector definition	p [i]
4	Vector expression	p [i] = NIL

4-7	Vector IF	IF p [i] = NIL THEN
		r [i].f := r [1].g
		ELSE
		r [i].f := r [i].f * p [i]^

5	Constant	1
5	Invariant	r [1].g
5	Vector definition	r [i].f
5	Vector assignment	r [i].f := r [1].g
7	Vector definition	p [i]^
7	Vector definition	r [i].f
7	Vector expression	r [i].f * p [i]^
7	Vector assignment	r [i].f := r [i].f * p [i]^

NOTE

FOR loops that contain IF statements, array-valued subscripts (indirect addressing of arrays), and array-valued pointer dereferencing vectorize only if the CIGS attribute (specifying compressed index and gather/scatter) is present on the CPU parameter of the Pascal invocation statement, either explicitly or as a default attribute of the target machine.

If the VI directive is present in a comment prior to a FOR statement, Pascal ignores ambiguous subscripts and other potential dependencies when analyzing the loop.

The examples of vectorizable FOR loops that follow assume these declarations:

```
CONST
   n = 1000;

VAR
   a, b: ARRAY [1..n] OF REAL;
   c: ARRAY [1..10] OF 1..n;
   d: ARRAY [1..n] OF
         PACKED RECORD
           f: ^ ARRAY [1..10] OF INTEGER;
           g: I24
         END;
   dot: REAL;

BEGIN
```

Example 1, a vectorizable FOR loop:

```
FOR i := 1 TO n DO BEGIN
      a [i] := 0.0;
      b [i] := SQR (b [i]) + 1
END;
```

Example 2, a FOR loop that shifts array elements:

```
FOR i := 1 TO n - 1 DO
      a [i] := a [i + 1];
```

Example 3, a FOR loop that produces the dot product of two vectors:

```
dot := 0.0;
FOR i := 1 TO n DO
      dot := dot + a [i] * b [i];
```

Example 4, complex addressing:

```
FOR i := 1 TO n DO
      d [i].f^[1] := d [i].g;
```

Example 5, indirect indexing that requires gather/scatter hardware for vectorization:

```
FOR i := 1 TO 10 DO
      a [c [i]] := b [c [i]];
```

Example 6, safe division:

```
FOR i := 1 TO n DO
      IF a [i] <> 0 THEN
         a [i] := 1 / a [i];
```

Example 7, an IF statement that requires gather/scatter hardware for vectorization:

```
FOR i := 1 TO n DO
      IF a [i] > b [i] THEN
       b [i] := 0
      ELSE
       a [i] := b [i];
```

Example 8, the program sieve, is an example of a program with several vectorizable loops. Another version of this program that uses array processing operations instead of FOR loops appears in section 6, Array Processing:

```
PROGRAM sieve (OUTPUT);

   CONST
         size = 1000000;

   TYPE
         flagarray = array [2..size] OF BOOLEAN;

   VAR
         flags: flagarray;
         i,j: integer;

   BEGIN
         FOR i := 2 TO size DO
          flags[i] := TRUE;
         FOR i:= 2 TO TRUNC(SQRT(size)) DO
          IF flags[i] THEN
            FOR j := i+i TO size BY i DO
              flags[j] := FALSE;
         j:=0;
         FOR i := 2 TO size DO
          j := j + ORD (flags[i]);
         WRITELN (' End vectorized Pascal sieve; ',
               j:0, ' Primes found. ')
         END.
```

13.2 OPTIMIZATION

High-level optimizations are performed before the code is generated for a Pascal program. The following types of optimizations are considered high-level optimizations:

- Propagation of variable definitions
- Common subexpression elimination
- Compile-time expression evaluation
- Register residency prediction
- Loop invariant expression detection

Low-level optimizations are performed after the code has been generated. The following types of optimizations are considered low-level optimizations:

- Reordering of instructions (scheduling)
- Dead instruction elimination
- Register transfer elimination

All optimizations are controlled by the O compiler option. Optimizations are enabled by O+ and disabled by O-. Since optimizations decrease execution time while increasing compilation time, use O- while developing a program and O+ for production runs.

The O compiler option can be followed with a number to fine-tune the instruction scheduler; the O+ option is equivalent to the O3 option. The number determines how much time the scheduler uses in its search for the optimal instruction schedule. The number specified has a direct affect on the time required for compilation with large numbers increasing compilation time and small numbers decreasing compilation time.

Although small numbers decrease compilation time, specifying a number that is too small can cause poorly scheduled code. The effect of instruction scheduling varies from one Pascal program to another. Experimenting with various numerical values in your program may be necessary to produce the best medium between compilation and execution time.

APPENDIX SECTION

A. CHARACTER SET

The ASCII character set contains the 128 control and graphic characters shown in table A-1.

The letters that appear in parentheses following the descriptions in the fifth column indicate the following control character usage.

Usage	Description
CC	Communication control
FE	Format effector
IS	Information separator

Table A-1. ASCII Character Set

Character	Octal Code	Punched-card Code	Decimal Code	Description
NUL	000	12-0-9-8-1	0	Null
SOH	001	12-9-1	1	Start of heading (CC)
STX	002	12-9-2	2	Start of text (CC)
ETX	003	12-9-3	3	End of text (CC)
EOT	004	9-7	4	End of transmission (CC)
ENQ	005	0-9-8-5	5	Enquiry (CC)
ACK	006	0-9-8-6	6	Acknowledge (CC)
BEL	007	0-9-8-7	7	Bell (audible or attention signal
BS	010	11-9-6	8	Backspace (FE)
HT	011	12-9-5	9	Horizontal tabulation (FE)
LF	012	0-9-5	10	Line feed (FE)

Character	Octal Code	Punched-Card Code	Decimal Code	Description
VT	013	12-9-8-3	11	Vertical tabulation (FE)
FF	014	12-9-8-4	12	Form feed (FE)
CR	015	12-9-8-5	13	Carriage return (FE)
SO	016	12-9-8-6	14	Shift out
SI	017	12-9-8-7	15	Shift in
DLE	020	12-11-9-8-1	16	Data link escape (CC)
DC1	021	11-9-1	17	Device control 1
DC2	022	11-9-2	18	Device control 2
DC3	023	11-9-3	19	Device control 3
DC4	024	9-8-4	20	Device control 4 (stop)
NAK	025	9-8-5	21	Negative acknowledge (CC)
SYN	026	9-2	22	Synchronous idle (CC)
ETB	027	0-9-6	23	End of transmission block (CC)
CAN	030	11-9-8	24	Cancel
EM	031	11-9-8-1	25	End of medium
SUB	032	9-8-7	26	Substitute
ESC	033	9-9-7	27	Escape
FS	034	11-9-8-4	28	File separator (IS)
GS	035	11-9-8-5	29	Group separator (IS)
RS	036	11-9-8-6	30	Record separator (IS)
US	037	11-9-8-7	31	Unit separator (IS)
(Space)	040	(None)	32	Space (blank)

Character	Octal Code	Punched-card Code	Decimal Code	Description
!	041	12-8-7	33	Exclamation mark
"	042	8-7	34	Quotation marks (diaeresis)
#	043	8-3	35	Number sign
$	044	11-8-3	36	Dollar sign (currency symbol)
%	045	0-8-4	37	Percent
&	046	12	38	Ampersand
'	047	8-5	39	Apostrophe (single close quotation)
(	050	12-8-5	40	Opening (left) parenthesis
)	051	11-8-5	41	Closing (right) parenthesis
*	052	11-8-4	42	Asterisk
+	053	12-8-6	43	Plus
,	054	0-8-3	44	Comma (cedilla)
-	055	11	45	Minus (hyphen)
.	056	12-8-3	46	Period (decimal point)
/	057	0-1	47	Slant (slash, virgule)
0	060	0	48	Zero
1	061	1	49	One
2	062	2	50	Two
3	063	3	51	Three

Character	Octal Code	Punched-card Code	Decimal Code	Description
4	064	4	52	Four
5	065	5	53	Five
6	066	6	54	Six
7	067	7	55	Seven
8	070	8	56	Eight
9	071	9	57	Nine
:	072	8-2	58	Colon
;	073	11-8-6	59	Semicolon
<	074	12-8-4	60	Less than
=	075	8-6	61	Equal
>	076	0-8-6	62	Greater than
?	077	0-8-7	63	Question mark
@	100	8-4	64	Commercial at-sign
A	101	12-1	65	Uppercase letter
B	102	12-2	66	Uppercase letter
C	103	12-3	67	Uppercase letter
D	104	12-4	68	Uppercase letter
E	105	12-5	69	Uppercase letter
F	106	12-6	70	Uppercase letter
G	107	12-7	71	Uppercase letter
H	110	12-8	72	Uppercase letter
I	111	12-9	73	Uppercase letter

Character	Octal Code	Punched-card Code	Decimal Code	Description
J	112	11-1	74	Uppercase letter
K	113	11-2	75	Uppercase letter
L	114	11-3	76	Uppercase letter
M	115	11-4	77	Uppercase letter
N	116	11-5	78	Uppercase letter
O	117	11-6	79	Uppercase letter
P	120	11-7	80	Uppercase letter
Q	121	11-8	81	Uppercase letter
R	122	11-9	82	Uppercase letter
S	123	0-2	83	Uppercase letter
T	124	0-3	84	Uppercase letter
U	125	0-4	85	Uppercase letter
V	126	0-5	86	Uppercase letter
W	127	0-6	87	Uppercase letter
X	130	0-7	88	Uppercase letter
Y	131	0-8	89	Uppercase letter
Z	132	0-9	90	Uppercase letter
[	133	12-8-2	91	Opening (left) bracket
\	134	0-8-2	92	Reverse slant (backslash)
]	135	11-8-2	93	Closing (right) bracket
^	136	11-8-7	94	Circumflex
-	137	0-8-5	95	Underline

Character	Octal Code	Punched-card Code	Decimal Code	Description
`	140	8-1	96	Grave accent (single open quotation)
a	141	12-0-1	97	Lowercase letter
b	142	12-0-2	98	Lowercase letter
c	143	12-0-3	99	Lowercase letter
d	144	12-0-4	100	Lowercase letter
e	145	12-0-5	101	Lowercase letter
f	146	12-0-6	102	Lowercase letter
g	147	12-0-7	103	Lowercase letter
h	150	12-0-8	104	Lowercase letter
i	151	12-0-9	105	Lowercase letter
j	152	12-11-1	106	Lowercase letter
k	153	12-11-2	107	Lowercase letter
l	154	12-11-3	108	Lowercase letter
m	155	12-11-4	109	Lowercase letter
n	156	12-11-5	110	Lowercase letter
o	157	12-11-6	111	Lowercase letter
p	160	12-11-7	112	Lowercase letter
q	161	12-11-8	113	Lowercase letter
r	162	12-11-9	114	Lowercase letter
s	163	11-0-2	115	Lowercase letter
t	164	11-0-3	116	Lowercase letter
u	165	11-0-4	117	Lowercase letter

Character	Octal Code	Punched-card Code	Decimal Code	Description
v	166	11-0-5	118	Lowercase letter
w	167	11-0-6	119	Lowercase letter
x	170	11-0-7	120	Lowercase letter
y	171	11-0-8	121	Lowercase letter
z	172	11-0-9	122	Lowercase letter
{	173	12-0	123	Opening (left) brace
|	174	12-11	124	Vertical line
}	175	11-0	125	Closing (right) brace
~	176	11-0-1	126	Overline (tilde, general accent)
DEL	177	12-9-7	127	Delete

B. PREDEFINED FUNCTIONS AND PROCEDURES

Table B-1 lists in alphabetic order predefined functions and procedures available to a Pascal program executing on a CRAY-2, CRAY X-MP, or CRAY-1 Computer System. These functions and procedures are specified in the ISO Level 1 Pascal standard. For more information on the mathematical functions for CRAY X-MP and CRAY-1 Computer Systems using Cray Pascal, see the equivalent routines in the Programmer's Library Reference Manual. For more information about the mathematical functions for CRAY-2 Computer Systems using Cray Pascal, see the equivalent routines in the CRAY-2 UNICOS Libraries, Macros, and Opdefs Reference Manual.

Table B-1 gives the subprogram name and parameters, indicates whether the subprogram is a function (F) or procedure (P), specifies whether or not the subprogram is in-line, and describes what the subprogram does.

Table B-1. Predefined Functions and Procedures

Call	F or P	In-line	Description
ABS(x)	F	No	Returns the absolute value of the integer or real number x. The result is of the same type as x.
ARCTAN(x)	F	No	Returns the arctangent, a real number, of the integer or real number x.
CHR(x)	F	Yes	Returns the ASCII character corresponding to the ordinal number x, which is an integer. A run-time error results if x is less than 0 or greater than 127.
COS(x)	F	No	Calculates the cosine of the integer or real number x. x must be less than 2^{24}.

Table B-1. Predefined Procedures and Functions (continued)

Call	F or P	In-line	Description
DISPOSE(*x*)	P	No	Deallocates the dynamic variable pointed to by *x*. *x* retains the same value, but *x*^ becomes undefined.
EOF(*x*)	F	No	Returns a value of TRUE if end-of-file is reached in file *x* and FALSE if it is not. The predefined file INPUT is the default if *x* is not specified.
EOLN(*x*)	F	No	Returns a value of TRUE if end-of-line is reached in file *x* and FALSE if it is not. The predefined file INPUT is the default if *x* is not specified.
EXP(*x*)	F	No	Raises *e* to a power of *x* (*e* = 2.718218). *x* is either an integer or real number, and the result is always a real number. *x* must be no greater than 5676.
GET(*x*)	P	No	Moves the window to the next element of file *x* and assigns the value of that element to the buffer variable. If the next element is the end-of-file, the buffer variable is undefined and the end-of-file condition is set to TRUE.
LN(*x*)	F	No	Determines the log base (*e*) of *x*. *x* is either an integer or real number greater than 0, and the result is always a real number.
NEW(*x*)	P	No	Allocates a dynamic variable pointed to by *x*, which must be previously declared as a pointer variable. The dynamic variable is of the type to which *x* is bound. (Section 11, Dynamic Allocation, describes a second form of the NEW procedure for dynamically allocating variant records.)

Table B-1. Predefined Procedures and Functions (continued)

Call	F or P	In-line	Description
ODD(*x*)	F	Yes	Returns a value of TRUE if *x* is an odd integer and FALSE if it is not
ORD(*x*)	F	Yes	Returns the ordinal number of *x*. *x* can be a variable of any scalar type except REAL.
PACK(*a,i,x*)	P	Yes	Copies elements from unpacked array *a* into packed array *x*, starting at element *i* of *a*
PAGE(*x*)	P	No	Causes the printer to skip to the top of the next page when printing text file *x*. If *x* is not specified, the predefined file OUTPUT is the default.
PRED(*x*)	F	Yes	Returns the element that precedes *x* in an ordered type. *x* cannot be the first element in the type.
PUT(*x*)	P	No	Writes the contents of the buffer variable to the window position at the end of file *x*. The end-of-file condition must be TRUE before PUT is executed.
RESET(*x*)	P	No	Sets the window at the beginning of file *x* before reading from that file. Unless the file is empty, the end-of-file condition is set to FALSE and the value of the first element in the file is assigned to the buffer variable.
REWRITE(*x*)	P	No	Deletes the contents of file *x* before writing to that file and sets the end-of-file condition to TRUE
ROUND(*x*)	F	No† Yes††	Rounds the real number *x* to the nearest integer value

† CRAY X-MP and CRAY-1 Computer Systems
†† CRAY-2 Computer Systems

Table B-1. Predefined Procedures and Functions (continued)

Call	F or P	In-line	Description
SIN(*x*)	F	No	Calculates the sine of the integer or real number *x*, which must be less than 2^{24}
SQR(*x*)	F	No	Returns the square of the integer or real number *x*
SQRT(*x*)	F	No† Yes††	Returns the square root of the integer or real number *x*, which must be greater than or equal to 0
SUCC(*x*)	F	Yes	Returns the element that follows *x* in an ordered type. *x* cannot be the last element in the type.
TRUNC(*x*)	F	No† Yes††	Truncates the fractional part of the real number *x*, returning it as an integer
UNPACK(*x*,*a*,*i*)	P	Yes	Copies elements from packed array *x* into unpacked array *a*, starting at element *i* of a

† CRAY X-MP and CRAY-1 Computer Systems
†† CRAY-2 Computer Systems

The predefined functions and procedures in table B-2 execute on CRAY-2, CRAY X-MP, and CRAY-1 Computer Systems and are CRI extensions to those of the ISO Level 1 Pascal standard. In these functions and procedures, *x* and *y* represent operands of type INTEGER, type I24, or type I32, *r* is an operand of type REAL, *f* is a file variable, *dn* is a dataset name, and *p* is a pointer variable.

Table B-2. Extensions to the Predefined Functions and Procedures

Call	F or P	In-line	Description
ALL(*x*)	F	Yes	Returns TRUE if any array element is TRUE

Table B-2. Extensions to the Predefined Functions and Procedures (continued)

Call	F or P	In-line	Description
ANY(*x*)	F	Yes	Returns TRUE if all of the array elements are TRUE
ARCCOS(*r*)	F	No	Calculates the inverse of cosine *r*
ARCSIN(*r*)	F	No	Calculates the inverse of sine *r*
BAND(*x,y*)	F	Yes	Determines the logical product (AND) of *x* and *y*
BNOT(*x*)	F	Yes	Determines the logical ones complement of *x*
BOR(*x,y*)	F	Yes	Determines the logical inclusive OR of *x* and *y*
BXOR(*x,y*)	F	Yes	Determines the logical exclusive OR of *x* and *y*
CONNECT(*f,dn*)	P	No	Associates the Pascal file variable *f* with the dataset *dn* (see section 10, Input and Output)
COSH(*r*)	F	No	Calculates the hyperbolic cosine of *r*
HALT	P	No	Terminates the execution of the Pascal program and generates a walkback listing (see appendix F, Debug Information, for debugging purposes
LOC(*x*)	F	Yes	Returns the address of *x*, which must be passed as a VAR parameter. This address is type-compatible with pointers. LOC defeats strong data typing and inhibits assignment of any user variables to B and T registers for the entire compile unit.
LOG(*x*)	F	Yes	Calculates the exponent of *x*

Table B-2. Extensions to the Predefined Functions and Procedures (continued)

Call	F or P	In-line	Description
LSHIFT(*x,y*)	F	Yes	Shifts the word *x* left *y* bit positions. Bits shifted off the left end are lost, and bit positions on the right are zero-filled.
MAXVAL(*x*)	F	Yes	Returns the largest element of array *x*
MINVAL(*x*)	F	Yes	Returns the smallest element of array *x*
POP(*x*)	F	Yes	Returns the population count (number of bits set to 1) in the word *x*
PRODUCT(*x*)	F	Yes	Computes the product of all of the elements in array *x*
RSHIFT(*x,y*)	F	Yes	Shifts the word *x* right *y* bit positions. Bits shifted off the right end are lost, and bit positions on the left are zero-filled.
SINH(*r*)	F	No	Calculates the hyperbolic sine of *r*
SIZEOF(*p*, *tag-field-value*$_1$, *tag-field-value*$_2$, . . . *tag-field-value*$_n$)	F	No	Returns the size (in Cray words) of a dynamic variable: that is, the integer number of words that a call to NEW will allocate when called with the same parameter list. SIZEOF takes a variable number of parameters and behaves exactly like NEW used as a function. The value of pointer *p* is returned unaltered.
SUM(*x*)	F	Yes	Computes the sum of all of the elements in array *x*
TAN(*r*)	F	No	Calculates the tangent of *r*
TANH(*r*)	F	No	Calculates the hyperbolic tangent of *r*

C. COMPILER ERROR MESSAGES

The Pascal compiler issues the following error messages during the compile pass. The generation of code is prevented by errors that cause any of these messages to be issued.

C.1 LISTING MESSAGES

2: Identifier expected
An identifier might be missing or misplaced. Check the syntax of the statement indicated and supply an identifier in the correct place.

3: 'PROGRAM' OR 'MODULE' EXPECTED
The first symbol of a Pascal program must be either PROGRAM or MODULE. (Comments are not symbols, so they can precede the PROGRAM or MODULE.) Change and recompile the program.

4: ')' EXPECTED
A right parenthesis might be missing or an extra left parenthesis might be present. Each left parenthesis must be balanced by a right parenthesis.

5: ':' EXPECTED
A colon might be missing. Check the syntax for the statement indicated and insert a colon.

6: INVALID SYMBOL
The program might include an invalid or missing symbol. Check that the statement indicated is separated from the previous statement by a semicolon. Also check the syntax of the statement to ensure that no reserved words are missing.

7: ERROR IN PARAMETER LIST
The parameter list contains an error. Check to see that the number of actual parameters agrees with the number of formal parameters and that the corresponding actual and formal parameters are of the same type. (See section 9, Procedures and Functions, for more information on parameter passing.)

8: 'OF' EXPECTED
The reserved word OF might be missing from the indicated line. Check the syntax of the statement involved and change the program accordingly.

9: '(' EXPECTED
A left parenthesis might be missing or an extra right parenthesis might be present. Each right parenthesis must be balanced by a left parenthesis.

11: '[' EXPECTED
A left bracket might be missing. The left bracket is used to introduce a subrange, an array index range, or a set. Check the syntax of the indicated statement and change the program accordingly.

12: ']' EXPECTED
A right bracket might be missing. The right bracket is used to close a subrange, an array index range, or a set. Check the syntax of the indicated statement and change the program accordingly.

13: 'END' EXPECTED
The reserved word END might be missing. An END must appear to balance each BEGIN. Check the program for an equal number of BEGIN and END reserved words and change it accordingly.

14: ';' EXPECTED
The program might be missing a semicolon. A semicolon is a separator between statements. Check the program to see if a semicolon is needed to separate run-together statements.

15: INTEGER EXPECTED
The data indicated should be of type INTEGER. The ISO Level 1 Pascal Standard specifies that labels must be integers. Check the program to make sure that all labels are integers, and that there are no missing or extraneous symbols.

16: '=' EXPECTED
An equal sign might be missing. Check the syntax of the declaration indicated and make the necessary change to the program. (Appendix E, Pascal Syntax, contains the complete Pascal syntax.)

17: 'BEGIN' EXPECTED
The reserved word BEGIN might be missing. BEGIN must appear immediately before the first executable statement in the main program segment and all procedures and functions. It also appears at the beginning of a compound statement. Check the syntax of the area of the program indicated and make any necessary changes.

20: ',' EXPECTED
A comma might be missing. A comma separates most list items, such as variables of the same type in a VAR declaration, actual parameters in a procedure or function call, and input or output elements in a READ or WRITE statement. Check the syntax for the statement indicated.

22: 'BEGIN' OR PROCEDURE DECLARATION EXPECTED
A procedure or function declaration was followed by something other than another procedure or function declaration or the reserved word BEGIN. Check the procedure declaration for an equal number of BEGIN and END reserved words, or check for misplaced declarations following the procedure declaration.

23: ';' OR '.' EXPECTED
A procedure or function declaration in the outer level of a module did not end with a semicolon or period. Check the procedure declaration for an equal number of BEGIN and END reserved words.

24: '.' OR PROCEDURE DECLARATION EXPECTED
A procedure or function declaration in the outer level of a module was followed by something other than another procedure or function declaration or a period. Check the procedure declaration for an equal number of BEGIN and END reserved words, or check for misplaced declarations following the procedure declaration.

25: ARRAY PROCESSING NOT ALLOWED HERE
Array expressions, slices, array-valued subscripts, and array-valued base variables can only appear in assignment statements or as arguments to certain predefined functions. They can not appear in other contexts, such as actual arguments to user procedures or functions. Correct the program and recompile.

26: BASE VARIABLE MUST BE WHOLE
Array-valued subscript expressions cannot be applied to a base variable that is the result of an array-valued subscript expression or an array-valued base variable field or pointer access. Change the statement and recompile the program.

27: TOO MANY R^ DIRECTIVES
Ten R^ directives are specified without matching R* directives prior to the indicated R^ directive. Remove some R^ directives or insert some R* directives and recompile the program.

28: TOO MANY R* DIRECTIVES
The indicated R* directive does not have a matching R^ directive. Remove the R* directive or insert a R^ directive and recompile the program.

29: 'ELSE' expected
The constant subexpression following the reserved word THEN in a conditional expression was not followed by the reserved word ELSE as expected.

40: VALUE PART ONLY ALLOWED IN MAIN PROGRAM
The VALUE statement is allowed only in the main program, not in nested procedures or functions. Remove the statement and recompile the program.

41: TOO FEW VALUES SUPPLIED
An attempt was made to initialize a structured variable with a VALUE statement, but too few data values were supplied to fully initialize the variable. Change the statement and recompile the program.

42: TOO MANY VALUES SUPPLIED
An attempt was made to initialize a structured variable with a VALUE statement, but too many data values were supplied to initialize the variable. Change the statement and recompile the program.

43: ALREADY INITIALIZED
A variable was initialized twice in a VALUE statement. Variables must not be initialized more than once in a VALUE statement. Remove one of the initializations and recompile the program.

44: TYPE IS NEITHER ARRAY NOR RECORD
The VALUE statement syntax for initializing arrays or records was used with a variable that is neither. Check the syntax of VALUE statements carefully.

46: ERROR IN CONFORMANT ARRAY SCHEMA
An error was detected while trying to parse a conformant array schema. Check the syntax of the schema (see section 9, Procedures and Functions). This error can arise also if the reserved word ARRAY is used as the type of a formal parameter. Only the name of a previously declared type can be used as the type of a formal parameter.

47: VALUE OUT OF RANGE IN A VALUE STATEMENT
An out-of-range value was supplied to a variable in a VALUE statement. Correct the value and recompile the program.

48: IMPORTED OR COMMON DATA CAN'T BE INITIALIZED
The VALUE statement cannot be used to initialize imported or common data. Change and recompile the program.

51: ':=' EXPECTED
An assignment operator was expected. The equal sign alone is not the assignment operator. Check the statement indicated and make any necessary changes.

52: 'THEN' EXPECTED
The reserved word THEN was not found in the IF statement. The syntax of the IF statement is as follows:

```
if-statement = "if" boolean-expression "then" statement [ else-part ] .
```

Change the statement and recompile the program.

53: 'UNTIL' EXPECTED
The keyword UNTIL might be missing from a REPEAT statement. Section 8, Assignment Statement and Program Control Statements, describes the REPEAT statement and gives its syntax. Check the syntax of the statement indicated and make any necessary changes.

54: 'DO' EXPECTED
The reserved DO might be missing from a FOR, WHILE, or WITH statement. Check the syntax of the statement indicated and change as necessary.

55: 'TO'/'DOWNTO' EXPECTED
The reserved word TO or DOWNTO might be missing from a FOR statement. Section 8, Assignment Statement and Program Control Statements, describes the FOR statement and gives its syntax. Make any necessary changes and recompile the program.

57: 'FILE' EXPECTED
The reserved word FILE might be missing from the declaration indicated. Section 5, Data Types, describes the FILE type and gives examples of declarations. Check the syntax of the declaration and make any necessary changes.

58: ERROR IN FACTOR
An error was found while trying to parse a factor. Look for extraneous characters in the source line, a misspelled variable, or a binary operator with one of its operands missing.

59: ERROR IN VARIABLE
An error was found in the variable indicated. Check that the variable is spelled correctly and that it is separated from surrounding words and symbols by the proper delimiter.

60: ONLY INNERMOST DIMENSION MAY BE PACKED
The ISO Level 1 Pascal Standard specifies that only the innermost dimension of a conformant array can be packed. Change the declaration of the conformant array.

61: ERROR IN REAL CONSTANT: DIGIT EXPECTED
A real number is not correctly represented. A constant of type REAL must have at least one digit on each side of the decimal point or be represented in scientific notation. Section 3, Pascal Vocabulary, describes the correct presentation of numbers.

62: STRING CONSTANT MUST NOT EXCEED SOURCE LINE
An invalid string was detected, or an extra apostrophe was found. A string cannot cross line boundaries. An extra apostrophe on a line by itself can also produce this message.

63: INTEGER OR REAL CONSTANT EXCEEDS RANGE
An integer or real number is outside of the range of valid numbers. The largest valid integer on a Cray computer system is MAXINT, which has a value of $2^{64}-1$ (9,223,372,036,854,775,807 in decimal). The smallest value is -MAXINT. The largest valid real number in a Pascal program is 10^{2464}.

65: '..' EXPECTED
While parsing a conformant array schema, a different token was found when .. was expected. Check the syntax of the statement and correct the source program.

66: PARAMETERS BOUND TO SAME CONFORMANT ARRAY SCHEMA HAVE DIFFERENT TYPES
When calling a procedure with conformant array parameters, two arrays of different types were bound to the same conformant array schema. When two arrays are passed to a single schema, both arrays must be of the same data type. See the example in section 9, Procedures and Functions. Check and correct the source program.

67: INDEX TYPES NOT COMPATIBLE
A procedure with a conformant array parameter was passed an array whose index type was incompatible with that of the formal parameter. See section 8, Assignment Statement and Program Control Statements, for a definition of assignment compatibility. Check and correct the source program.

68: ACTUAL INDEX TYPE NOT SUBSET OF FORMAL INDEX TYPE
A procedure with a conformant array parameter was called with an actual parameter whose upper and lower bounds do not lie within the range of the conformant array parameter's index type. Change the conformant array schema to include the upper and lower bounds of the actual parameter.

69: CASE VALUE TOO LARGE OR TOO SMALL
Only values in the range -524287 to 524287 can be used in CASE statements. Change and recompile the program.

70: TOO MANY NESTED SCOPES
The program has procedures or WITH statements nested too deeply. The maximum depth of nested procedures or WITH statements is 20. Change and recompile the program.

71: TOO MANY NESTED PROCEDURES AND/OR FUNCTIONS
The maximum nesting depth for procedures and functions was exceeded. The maximum nesting depth is 25. Reorganize the program to decrease the nesting level of subprograms.

72: TOO MANY PROCEDURES
The limit for procedure definitions has been exceeded. The limit is 1,000. Reorganize the program to cut down the number of procedures.

73: PROCEDURE/PROGRAM TOO LONG
A procedure, function, or the main body of the Pascal program exceeds the maximum size for any of several possible reasons. Divide it into smaller segments and recompile the program.

75: TOO MANY ERRORS ON THIS SOURCE LINE
More than 10 errors were found on this source line; not all of the errors have been printed. Make the corrections suggested by the errors that have been printed.

79: TOO MANY EXPRESSIONS
The procedure or function contains too many expressions. The number of expressions in a procedure is limited and depends on the nature of the expressions. Split the procedure or function into several pieces and recompile the program.

80: DIVISION OR MOD BY 0
An attempt was made to DIV or MOD something by zero; this is mathematically meaningless. Change and recompile the program.

85: CONFORMANT ARRAY FORMAL PARAMETER MUST NOT BE PASSED BY VALUE
The program attempted to pass a conformant array formal parameter as a value parameter. Pass the conformant array formal parameter as a VAR parameter.

86: COMPONENT OF PACKED STRUCTURE MUST NOT BE PASSED AS VAR PARAMETER
The ISO Level 1 Pascal standard does not permit components of packed structures to be passed as VAR parameters. Components of packed structures can be passed as value parameters. Correct and recompile the program.

87: LOOP CONTROL VARIABLE ACTIVE IN ENCLOSING LOOP
The compiler encountered an attempt by nested looping structures to use the same variable as a loop control variable. An inner loop cannot use the same control variable as the loop surrounding it. Change the loop control variable for one of the structures and recompile the program.

93: UNRECOGNIZED COMPILER DIRECTIVE
A string in a comment beginning with a pound sign, which the compiler expected to be a compiler directive, was not a valid directive. If the string is supposed to be a directive, correct it; otherwise, add one or more spaces before it so it is ignored.

94: DIRECTIVE CANNOT BE USED WITHIN COMPILE UNIT
A compiler directive that can only be used before a PROGRAM or MODULE statement appeared within a compile unit. Move the directive outside of the compile unit or change it to a comment.

95: LABEL MUST BE ONE TO FOUR DIGITS
A label declaration included a label with more than 4 digits. Change the label to have 4 or fewer digits.

96: USE OF TYPE IDENTIFIER IN ITS DEFINITION
Use of a type name in the definition of that type is valid only for declaring a component of the type to be a pointer to the type. Change and recompile the program.

97: NOT ISO STANDARD
The indicated statement is not supported under the ISO Level 1 Pascal standard. If this program is to be run on a compiler that supports no more than the ISO standard Pascal, the statement must be changed.

101: IDENTIFIER DECLARED TWICE
The same identifier was declared twice in the same program block. Remove one of the declarations or rename one of the identifiers.

102: LOW BOUND EXCEEDS HIGH BOUND
The lower end of the specified range was given a larger value than the upper end of the range. Change the range specification to give a larger value to the upper bound or a lower value to the lower bound.

103: IDENTIFIER IS NOT OF APPROPRIATE CLASS
An identifier was used in an inappropriate context. For example, a constant was used as the target of an assignment statement. Change the statement to employ the correct kind of identifier.

104: IDENTIFIER NOT DECLARED
The identifier indicated might not have been declared in the declarations section of the program, procedure, or function. If the identifier was declared, check to see that it is spelled the same way in the executable statement as in the declaration.

105: SIGN NOT ALLOWED
A plus or minus sign was used with a constant of the wrong sort; for example, with a string constant. Remove the sign.

106: NUMBER EXPECTED
A number might be missing at the position indicated. Check the syntax of the statement to see if a number should be specified. Also, check the statement for a missing reserved word.

107: INCOMPATIBLE SUBRANGE TYPE
An attempt was made to define a subrange whose upper and lower limits were not of the same type, or were not elements of a scalar type other than real. Change the program accordingly and recompile it.

108: FILE NOT ALLOWED HERE
The data type FILE is not valid in the statement indicated. Change the statement to avoid the invalid specification and recompile the program.

110: TAGFIELD TYPE MUST BE SCALAR OR SUBRANGE
The data type of the tagfield in a variant record might be incorrect. The tagfield must either be a scalar or subrange type. Change the statement and recompile the program.

111: INCOMPATIBLE WITH TAGFIELD TYPE
A record variant selector is incompatible with the type of the tagfield. Correct and run the program.

112: INDEX TYPE MUST NOT BE REAL
The data type REAL was specified for an index. REAL is not a valid index type. An index type can be any scalar type, or a subrange of any scalar type, except REAL. Change the statement and recompile the program.

113: INDEX TYPE MUST BE SCALAR OR SUBRANGE
An invalid data type was specified for an index. An index type can be any scalar type, or a subrange of any scalar type, except REAL. Change the statement and recompile the program.

114: BASE TYPE MUST NOT BE REAL
The base type specified when defining a set was REAL. REAL is not a valid base type. The base type can be BOOLEAN, an enumerated type of up to 128 elements, a subrange of type INTEGER with a maximum range of 0 through 127, or a subrange of type CHAR. Change the SET declaration and recompile the program.

115: BASE TYPE MUST BE SCALAR OR SUBRANGE
An invalid base type was specified when defining a set. The base type can be BOOLEAN, an enumerated type of up to 128 elements, a subrange of type INTEGER with a maximum range of 0 through 127, or a subrange of type CHAR. Change the SET declaration and recompile the program.

116: ERROR IN TYPE OF STANDARD PROCEDURE PARAMETER
The actual parameter passed to a standard procedure was of an invalid type. If the actual parameter was passed as a value parameter, its assignment must be compatible with the corresponding formal parameter. (Section 8, Assignment Statement and Program Control Statements, describes assignment compatibility.) If the actual parameter is passed as a VAR parameter, it must be of the same type as the corresponding formal parameter. Correct and recompile the program.

117: UNSATISFIED FORWARD REFERENCE
A FORWARD directive referenced a procedure or function that was not found. Check that the spelling of the procedure or function is the same in the heading that includes the FORWARD directive and in the actual procedure or function heading.

118: UNSATISFIED FORWARD REFERENCE TYPE IDENTIFIER
One type was declared to be a pointer to another type, but the other type was not declared. Declare the type in a TYPE statement and recompile the program.

119: FORWARD DECLARED; REPETITION OF PARAMETER LIST NOT ALLOWED
The parameter list for a procedure or function referenced by a FORWARD directive appeared in the actual procedure or function heading. The parameter list is allowed only with the heading containing the FORWARD directive. Remove the parameter list from the actual heading.

120: FUNCTION RESULT TYPE MUST BE SCALAR, SUBRANGE, OR POINTER
A function was declared with an invalid result type. The valid types for a function are INTEGER, I24, REAL, BOOLEAN, CHAR, an enumerated type, a subrange, and pointer. Change the result type in the function heading and the function itself to include one of these types.

121: FILE VALUE PARAMETER NOT ALLOWED
The program attempted to pass a file as a value parameter. A file can only be passed as a VAR parameter. Change the procedure or function heading to make the file a VAR parameter.

122: FORWARD DECLARED FUNCTION: REPETITION OF RESULT TYPE NOT ALLOWED
The result type for a function referenced by a FORWARD directive appeared in the actual function heading. The result type is allowed only with the heading containing the FORWARD directive. Remove the result type from the actual heading.

124: FLOATING-POINT FORMAT FOR REALS ONLY
A WRITE or WRITELN statement applied an invalid format specification to a nonreal variable or expression. An output format for a nonreal variable or expression cannot contain a specification for the number of decimal places. The syntax for an output specification for a nonreal variable or expression is as follows:

```
element = 'constant' | expression [ ":" field-width ] .
```

125: ERROR IN TYPE OF STANDARD FUNCTION PARAMETER
The actual parameter passed to a standard function is of an invalid type. If the actual parameter is passed as a value parameter, its assignment must be compatible with the corresponding formal parameter. (Section 8, Assignment Statement and Program Control Statements, describes assignment compatibility.) If the actual parameter was passed as a VAR parameter, it must be of the same type as the corresponding formal parameter. Correct and recompile the program.

126: NUMBER OF PARAMETERS DOES NOT AGREE WITH DECLARATION
The number of actual parameter passed to a procedure or function was not the same as the number of formal parameters in the subprogram declaration. Check the procedure or function declaration and change the actual parameter list.

127: FUNCTION IS NOT A VALID CONSTANT FUNCTION
A function used in a constant definition cannot be used in a constant expression. Check the list of functions that can be used in constant expressions.

128: RESULT TYPE OF PARAMETER FUNCTION DOES NOT AGREE WITH DECLARATION
A function that appears as an actual parameter has a different result type from the corresponding formal function parameter. Match the types and recompile the program.

129: TYPE CONFLICT OF OPERANDS
Incompatible operands were detected. Operands of the same operation must be assignment compatible. Section 8, Assignment Statement and Program Control Statements, defines assignment compatibility. Change and recompile the program.

131: TESTS ON EQUALITY ALLOWED ONLY
The program attempted an invalid comparison of pointers. Pointers can be tested for equality but not for relative size (<, >, <=, and >=). Change and recompile the program.

132: STRICT INCLUSION NOT ALLOWED
The program attempted to determine if a set is strictly included within another set (that is, if one set is a proper subset of another). The ISO Level 1 Pascal standard permits tests for inclusion or equality (<=) but not for strict inclusion (<). Change and recompile the program.

133: FILE COMPARISON NOT ALLOWED
A relational operator (<, >, =, <=, >=, <>, and IN) was used with one or more operands of type FILE. Such an operation is not valid. Change and recompile the program.

134: INVALID TYPE OF OPERAND(S)
One or more operands of an invalid data type were found in an operation. Section 5, Data Types, describes the data types valid for each operation. Change and recompile the program.

135: TYPE OF OPERAND MUST BE BOOLEAN
The indicated operand must be of type BOOLEAN. The operators AND and OR require two Boolean operands, and the operator NOT requires a single Boolean operand. Change the operation and recompile the program.

136: SET ELEMENT TYPE MUST BE SCALAR OR SUBRANGE
An attempt was made to use the IN operator with an operand whose type is not a valid set base type. The base type of a set must be one of the following: an enumerated type, a subrange of types INTEGER or I24, or a subrange of type CHAR. Check the declaration of the first operand of the IN operator to be sure that it is of one of those types, and recompile the program.

137: SET ELEMENT TYPES NOT COMPATIBLE
An element of an incompatible data type was used with a set. Set element types are compatible if they are the same type, if one is a subrange of the other, or if both are subranges of the same type. In addition, either both sets must be packed or neither must be packed. Change and recompile the program.

138: TYPE OF VARIABLE IS NOT ARRAY
The program attempted to treat as an array a variable that was not declared as an array. Check the declaration of the variable and the statement indicated for inconsistencies.

139: INDEX TYPE IS NOT COMPATIBLE WITH DECLARATION
An index type other than the type specified in the declaration was used with a structured data type. Check the declaration of the variable and the statement indicated for inconsistencies.

140: TYPE OF VARIABLE IS NOT RECORD
The program attempted to treat as a record a variable that was not declared as a record. Check the declaration of the variable and the statement indicated for inconsistencies.

141: TYPE OF VARIABLE MUST BE FILE OR POINTER
The indicated variable must be of type FILE or POINTER. Change the operation and recompile the program.

142: INVALID PARAMETER SUBSTITUTION
A formal parameter was passed an invalid actual parameter value. Compare the positions of the formal parameters with the positions of the actual parameters for inconsistencies. Ensure that the parameters have been properly declared. Change and recompile the program.

143: INVALID TYPE OF LOOP CONTROL VARIABLE
A loop control variable of an invalid type was discovered. The control variable in a FOR loop must be of an ordinal type: that is, any scalar type except REAL. The beginning and final values of the control variable must be assignment-compatible with its type. Section 8, Assignment Statement and Program Control Statements, describes assignment compatibility and the FOR loop.

144: INVALID TYPE OF EXPRESSION
An expression resolving to an invalid type was discovered. The expression must resolve to a value that is assignment compatible with the variable to which it is assigned. (Section 8, Assignment Statement and Program Control Statements, describes assignment compatibility.)

145: TYPE CONFLICT
A data type conflict was discovered. Check the description of assignment compatibility and type compatibility in section 8, Assignment Statement and Program Control Statements, and change the program.

146: ASSIGNMENT OF FILES NOT ALLOWED
A file name appeared as an operand in an assignment statement. A file cannot be accessed directly in an assignment statement. The predefined procedures READ, READLN, WRITE, WRITELN, GET, and PUT must be used to move data to and from files. The buffer variable for a file can appear, however, in an assignment statement. The form of a buffer variable is as follows:

```
buffer-var = file-var "^" .
```

147: LABEL TYPE INCOMPATIBLE WITH SELECTING EXPRESSION
The data type of a label in a CASE statement or a record variant is incompatible with the data type of the selector. Make sure the labels are all of the same type. Also, check the declaration of any variables in the selector expression and ensure that the selector resolves to a constant of the same type as the labels.

148: SUBRANGE BOUNDS MUST BE SCALAR
The constants delimiting the range of a subrange are of an invalid type. The constants specifying the subrange bounds must be elements in any previously defined scalar type. Change the subrange specification and recompile the program.

151: ASSIGNMENT TO FORMAL FUNCTION NOT ALLOWED
The program attempted to assign a value to a formal function parameter. Such an assignment is not valid. Check the spelling of the assignment target and change the statement.

152: NO SUCH FIELD IN THIS RECORD
The program referred to a field not named in the record declaration. Check the spelling of the field name in the statement specified and in the declaration for consistency. Correct and recompile the program.

155: CONTROL VARIABLE MUST BE NEITHER FORMAL NOR NONLOCAL
An invalid use or declaration of the control variable was discovered. The control variable for a FOR statement must be declared in the declarations section of the block in which it is used. In addition, the control variable cannot be the target of an assignment statement, be passed to a subprogram as a VAR parameter, appear as a parameter in a READ or READLN statement, or be the control variable in more than one FOR statement. Change the invalid use of the control variable as required.

156: MULTIDEFINED CASE LABEL
The same label was defined more than once in a CASE statement or in a record variant. Each label must be unique. Change the label and recompile the program.

157: TOO MANY CASES IN CASE STATEMENT
More than 600 case constants were supplied in a single CASE statement. Break the CASE statement into pieces and recompile the program.

158: MISSING CORRESPONDING VARIANT DECLARATION
In a call to the predefined procedure NEW or SIZEOF, a case constant was supplied that did not correspond to a variant of the record type in question. Check the call and record type for consistency, or check if the call to NEW or SIZEOF contains too many parameters.

160: PREVIOUS DECLARATION WAS NOT FORWARD
A procedure was found with a name that is already a valid identifier in the current scope. Check the spelling of the procedure name.

162: PARAMETER SIZE MUST BE CONSTANT
In a call to the predefined procedure NEW or SIZEOF, a case constant was supplied when the record type in question had no variants. Remove the case constant from the call and recompile the program.

165: MULTIDEFINED LABEL
The same statement label appeared in more than one place. No statement label can appear more than once in a Pascal program. Change the label and recompile the program.

166: MULTIDECLARED LABEL
The same number was declared as a statement label more than once. No statement label can be declared more than once in a Pascal program. Change the label to a different number.

167: UNDECLARED LABEL
A statement label that was not declared appeared in the executable statements section of a program. Each statement label must be declared in a LABEL declaration in the declarations section of the block in which it is used. Section 3, Pascal Vocabulary, describes statement labels.

168: UNDEFINED LABEL
A statement label was specified in a LABEL declaration but was not used in the executable statements section of the block. Each statement label declared must be used. Remove the label from the LABEL declaration if it is not needed.

171: PREDEFINED FILE WAS REDECLARED
A predefined file (either INPUT, OUTPUT, or both) was specified in a declaration. Predefined files cannot be declared in a program. Remove the declaration and recompile the program.

172: UNDECLARED EXTERNAL FILE
The program attempted to access a file that was not declared. Except for the predefined files INPUT and OUTPUT, all files must be declared in the declarations section of the block in which they are used. Add the file declaration and recompile the program.

175: MISSING FILE 'INPUT' IN PROGRAM HEADING
The predefined text file INPUT was accessed in the program without being specified as a parameter in the PROGRAM heading. Add INPUT to the PROGRAM heading and recompile the program.

176: MISSING FILE 'OUTPUT' IN PROGRAM HEADING
The predefined text file OUTPUT was accessed in the program without being specified as a parameter in the PROGRAM heading. Add OUTPUT to the PROGRAM heading and recompile the program.

177: '.' EXPECTED
A period was expected. Check the indicated statement and make any necessary changes.

178: EXTERNAL FILE NAME MUST BE 7 CHARACTERS OR LESS
The compiler encountered a program parameter (a parameter in the PROGRAM declaration) longer than 7 characters. Change and recompile the program.

179: LABEL SECTION NOT ALLOWED HERE
An attempt was made to declare a label in an invalid place. Change the program and recompile the program.

184: CAN NOT EXPORT NESTED ROUTINES
Only routines declared at the outermost level of a program or module can be exported. Change the program and recompile the program.

185: COMPILER ERROR; SEND LISTING TO CRAY SUPPORT
An error has occurred in the execution of the Pascal compiler. Show a program listing to a CRI analyst.

186: NO CASE LIST ELEMENTS IN CASE STATEMENT
A CASE statement was encountered with no case list elements that is, with no case alternatives and corresponding statements. The ISO Level 1 Pascal standard does not permit this situation. Either add case list elements or remove the CASE statement.

187: SET ELEMENT OUT OF RANGE
The compiler encountered an element of a set whose value was less than 0 or greater than 127. Set elements must lie in the range of 0 through 127. Change and recompile the program.

188: BASE TYPE OF SET OUT OF RANGE
The compiler encountered a set TYPE declaration whose base type contained elements less than 0 or greater than 127. Set elements must lie in the range of 0 through 127. Change and recompile the program.

190: EXPORTED VARIABLE DECLARED TWICE
Two variables were exported with the same name; this is illegal. Change one of the names and recompile the program.

191: EXTERNAL VARIABLE NAME MUST BE 8 CHARACTERS OR LESS
The external names of exported variables cannot be longer than 8 characters. Shorten the name and recompile the program.

192: VIEWED VARIABLE NEW TYPE > OLD TYPE
An attempt was made to view a variable of a new type that uses more space than the old type. Variables can be viewed only with new types that are the same size as or smaller than the old type. Change and recompile the program.

193: CONFORMANT ARRAYS CAN NOT BE 'VIEWED'
Conformant arrays cannot be used in VIEWING statements. Change and recompile the program.

195: GOTO INTO CONTROL STRUCTURE
An attempt was made to jump into a structured statement, such as the THEN clause of an IF statement. Such a jump is illegal; change and recompile the program.

196: STRING CONSTANT TOO LONG
A string constant longer than 140 characters was declared. Change and recompile the program.

197: LOOP CONTROL VARIABLE CAN NOT BE PASSED AS VAR PARAMETER
Within a FOR loop, the loop control variable cannot be passed as a VAR parameter. Change and recompile the program.

198: READ INTO LOOP CONTROL VARIABLE NOT ALLOWED
Within a FOR loop, the loop control variable cannot appear as the target of a read. Change and recompile the program.

199: ASSIGMENT TO LOOP CONTROL VARIABLE NOT ALLOWED
Within a FOR loop, the loop control variable cannot appear as the target of an assignment. Change and recompile the program.

200: TOO MANY PARAMETERS
The procedure or function has too many parameters. The number of parameters permitted depends on their types. Change the procedure to use fewer parameters; either break it into smaller procedures or pass parameters implicitly by making nonlocal addresses.

201: ASSIGNMENT TO FUNCTION OUT OF SCOPE
An attempt was made to assign to a function result outside the scope of the function. Change and recompile the program.

202: ASSIGNMENT TO FUNCTION POINTEE NOT ALLOWED
A function result variable cannot be accessed on the left side of an assignment statement. Change the statement and recompile the program.

203: MULTIDEFINED VARIANT
The indicated variant value appears more than once within a record. Change the variant and recompile the program.

C.2 LOGFILE MESSAGES

When executing the Pascal compiler under UNICOS, all of the logfile messages go to standard error except the following: PS001, PS002, PS003, and PS005.

PS001 - [PASCAL] COMPILED *s*, *n* SOURCE LINES
The Pascal compiler has completed compilation of program module *s*, which consisted of *n* source lines. Class, informative.

PS002 - [PASCAL] CODE: *n* OCTAL, DATA: *m* OCTAL
The Pascal compiler generated *n* (octal) words of code and *m* (octal) words of static data for the current program module. Class, informative.

PS003 - [PASCAL] STACK: *n* OCTAL, HEAP: *m* OCTAL
At run time, the program compiled by the Pascal compiler initially requests *n* (octal) words of stack space and *m* (octal) words of heap space. Class, informative.

PS004 - [PASCAL] *n* ERRORS IN *s*, NO CODE GENERATED
Errors were detected by the Pascal compiler during the compilation of program module *s*; no code was generated. The Pascal compiler aborts the job step if the A+ option was on the compiler call line. Class, fatal.

PS005 - [PASCAL] NORMAL TERMINATION
The Pascal compiler terminated without errors. Class, informative.

PS006 - [PASCAL] SOURCE LINE TOO LONG, NO CODE GENERATED
The Pascal compiler detected a source line greater than 140 characters in width. Compilation was aborted. Class, fatal.

PS007 - [PASCAL] PREMATURE EOF ON INPUT SOURCE FILE
The Pascal compiler detected an end of file on the input source file prior to the end of the current program module. Compilation was aborted. Class, fatal.

PS008 - [PASCAL] PREMATURE EOD ON INPUT SOURCE FILE
The Pascal compiler detected an end of data on the input source file prior to the end of the current program module. Compilation was aborted. Class, fatal.

PS009 - [PASCAL] HARDWARE I/O ERROR ON INPUT SOURCE FILE
The Pascal compiler detected an unrecoverable hardware I/O error on the input source file. Compilation was aborted. Class, fatal.

PS010 - [PASCAL] UNRECOGNIZED KEYWORD *s* ON COMMAND LINE IGNORED
The Pascal compiler does not recognize a keyword (*s*) on the command line. Class, fatal.

PS011 - [PASCAL] INVALID INPUT FILE NAME *s*, COMPILE TERMINATED
The value specified for the I parameter on the Pascal call line is invalid. Class, fatal.

PS012 - [PASCAL] INVALID LIST FILE NAME *s*, COMPILE TERMINATED
The value specified for the L parameter on the Pascal call line is invalid. Class, fatal.

PS013 - [PASCAL] INVALID BLD FILE NAME *s*, COMPILE TERMINATED
The value specified for the B parameter on the Pascal call line is invalid. Class, fatal.

PS014 - [PASCAL] INPUT FILE SPECIFIED TWICE, COMPILE TERMINATED
The I parameter has occurred more than once on the Pascal call line. Class, fatal.

PS015 - [PASCAL] LIST FILE SPECIFIED TWICE, COMPILE TERMINATED
The L parameter has occurred more than once on the Pascal call line. Class, fatal.

PS016 - [PASCAL] BLD FILE SPECIFIED TWICE, COMPILE TERMINATED
The B parameter has occurred more than once on the Pascal call line. Class, fatal.

PS017 - [PASCAL] BLD FILE AND LIST FILE SAME, COMPILE TERMINATED
You specified the same dataset name for both the B and L parameters on the Pascal call line. Class, fatal.

PS018 - [PASCAL] INPUT FILE AND LIST FILE SAME, COMPILE TERMINATED
You specified the same dataset name for both the I and L parameters on the Pascal call line. Class, fatal.

PS019 - [PASCAL] INPUT FILE AND BLD FILE SAME, COMPILE TERMINATED
You specified the same dataset name for both the I and B parameters on the Pascal call line. Class, fatal.

PS020 - [PASCAL] ABNORMAL TERMINATION
The Pascal compiler is unable to successfully complete compilation of the current program module. Class, fatal.

PS021 - EXTENDED MODE RELOCATABLE GENERATED
The generated binary expects to be run in EMA (enhanced memory addressing) mode. Class, informative.

PS022 - UNRECOGNIZED OPTION IN CONTROL STATEMENT: *option*
A string specified with the O parameter in the PASCAL control statement was not a valid option. Class, fatal.

PS023 - CONTROL STATEMENT TOO LONG, COMPILE TERMINATED
The PASCAL control statement contained more than 300 nonblank characters. Class, fatal.

PS024 - CONTROL STATEMENT TERMINATOR MISSING
A continuation line of the PASCAL control statement did not end with a period or continuation character. Class, fatal.

PS025 - CONTROL STATEMENT CONTINUATION LINE NOT FOUND
The last line in the control statement file was a line from the PASCAL control statement that ended with a continuation character. Class, fatal.

D. RUN-TIME MESSAGES

The messages described in this appendix are issued by routines that are part of the Pascal run-time library. The message code indicates the type of routine issuing the message as follows:

Message Code	Function of Routine
RT10*nn*	Memory management or range checking
RT30*nn*	Input/output

All run-time messages identify fatal errors that cause your program to produce unpredictable results. The messages are as follows:

RT1000 - PROGRAM CALLED HALT
The program called the HALT procedure, which terminates the execution of the program. From P$HALT.

RT1001 - HEAP SPACE EXAUSTED
All of the memory space set aside for dynamic allocation has been used. From P$NEW.

RT1002 - DISPOSED AREA NOT IN HEAP
A DISPOSE statement attempted to deallocate a dynamic variable that was not currently allocated. Check the program logic to ensure that each DISPOSE statement is associated with a NEW statement to allocate the dynamic variable. From P$DISP.

RT1003 - DISPOSED AREA HAS BAD LINKAGE WORD
The heap management linkage word has been altered. Either the pointer being disposed has been stored indirectly with negative offset or the allocated area prior to the disposed area has been stored indirectly past its bounds. Turn on range checking, then recompile and rerun the program. From P$DISP.

RT1004 - INTEGER OVERFLOW
An integer in the program has exceeded the highest possible value. If the variable was declared as type I24 (a 24-bit integer), a larger value can be accommodated by changing the declaration to type INTEGER (64 bits). MAXINT specifies the largest 64-bit integer value possible on a machine. (The value of MAXINT on a Cray computer system is $2^{64} - 1$.) If MAXINT has been exceeded, the program must be modified. From any Pascal routine.

RT1005 - STACK OVERFLOW
All slots in the run-time stack are full as a result of too many recursive procedure or function calls. Check for the possibility of infinite recursion. To increase the stack size, reset the value of the S compiler directive (described in section 2, Using Pascal on a Cray Computer). From any Pascal routine.

RT1006 - DIVISION BY ZERO
The program attempted to divide a number by 0. Dividing by 0 is not a valid operation. Change the program to avoid division by 0. From any Pascal routine with the division by 0 checking option on.

RT1007 - NO CASE PROVIDED FOR THIS VALUE
None of the labels in a CASE statement matched the value of the selector, and the optional OTHERWISE clause was not specified. Add an OTHERWISE clause (described in section 8, Assignment Statement and Program Control Statements) to handle unmatched values. From any Pascal routine with the CASE statement checking option on.

RT1008 - INDEX EXPRESSION OUT OF BOUNDS
The program attempted to access an element outside of the bounds of the declared structure. Change the program to ensure that the index value does not exceed the declared bounds. From any Pascal routine with the index checking option on.

RT1009 - ASSIGNMENT OUT OF BOUNDS
In an assignment statement, the value on the right side was outside of range of legal values for the variable on the left side. For example, a value of -3 on the right side is out of bounds for a variable on the left side with a declared range of -2..4. Change the program to prevent such an assignment. From any Pascal routine compiled with the assignment checking option on.

RT1010 - INVALID POINTER REFERENCE
The program attempted to use a pointer variable that did not point to a valid item. Either the pointer was not initialized or the item to which it pointed had been deallocated (using the predefined procedure DISPOSE). Check these possible errors in the program and make appropriate changes. From any Pascal routine compiled with the pointer checking option on.

RT1011 - SUCC FUNCTION OUT OF BOUNDS
The SUCC function attempted to access an element outside of the declared bounds of the type. If the last element in the type, for instance, is specified as a parameter in the SUCC function call, the function attempts to access an invalid element. Change the program to ensure a valid parameter. From any Pascal routine with the SUCC function checking option on.

RT1012 - PRED FUNCTION OUT OF BOUNDS
The PRED function attempted to access an element outside of the declared bounds of the type. If the first element in the type, for instance, is specified as a parameter in the PRED function call, the function attempts to access an invalid element. Change the program to ensure a valid parameter. From any Pascal routine with the PRED function checking option on.

RT1013 - SET ELEMENT OUT OF BOUNDS
The program referenced an element that is outside of the bounds of the specified set. Change the program to ensure that only elements included in the set are referenced. From any Pascal routine with the SET checking option on.

RT1015 - HEAP CHECKING CAUGHT DAMAGED HEAP
The heap data structure (used internally for dynamic allocation) was corrupted. The point of corruption is indicated if the R+ option is specified on the Pascal invocation statement.

RT1016 - DISPOSE OF UNALLOCATED AREA
A Pascal DISPOSE statement attempted to deallocate a dynamic variable that was not currently allocated. Check the program to ensure that the argument to the DISPOSE procedure is a valid pointer. From P$DISP.

RT1017 - CHR FUNCTION ARGUMENT OUT OF BOUNDS
The CHR function was called with an argument less than 0 or greater than 127. Change and rerun the program. From any Pascal routine with the CHR function checking option set on.

RT1018 - MOD BY ZERO OR NEGATIVE
The program attempted to perform a MOD function with modulus less than or equal to 0. Change and rerun the program. From any Pascal routine with the MOD checking option turned on.

RT1019 - FOR INDEX INITIAL VALUE OUT OF BOUNDS
The initial value of the index variable in a FOR loop is not a legal value for that variable. Either change the variable's type or ensure that the initial value is legal. From any Pascal routine compiled with the range checking option on.

RT1020 - FOR INDEX FINAL VALUE OUT OF BOUNDS
The final value of the index variable in a FOR loop is not a legal value for that variable. Either change the variable's type or ensure that the final value is legal. From any Pascal routine compiled with the range checking option on.

RT1021 - REPRIEVE PROCESSING INITIATED
The program enabled reprieve processing with the P$REPRV library routine, and a reprievable error occurred. From P$REPRV.

RT1022 - WRONG VARIANT ACTIVE
A field was accessed when its variant was inactive. From any Pascal routine with the variant checking option set on.

RT1023 - VALUE PARAMETER OUT OF RANGE
A value parameter is out of range. From any Pascal routine with range checking set on.

RT1025 - UNCONFORMABLE ARRAYS

RT3001 - NAMED FILE DOES NOT EXIST
The file that was to be opened could not be located. Check the spelling of the file name and ensure that a file of the same name is local to the job. From P$RESET and P$REWRIT.

RT3004 - ATTEMPTED READ ON UNOPENED FILE
The file could not be read because it was not previously opened. Open the file using the RESET standard procedure before attempting the read. From any Pascal I/O routine.

RT3005 - ATTEMPTED WRITE TO AN UNOPENED FILE
The file could not be written because it was not previously opened. Open the file using the REWRITE standard procedure before attempting the write. From any Pascal I/O routine.

RT3006 - FUNCTION ATTEMPTED ON UNOPENED FILE
The file could not be used because it was not previously opened. Open the file using the RESET or REWRITE standard procedures before attempting the function. From any Pascal I/O routine.

RT3007 - ATTEMPTED READ PAST EOF/EOD
The program attempted to read beyond the end-of-file or the end-of-dataset indicator. Ensure that EOF is not TRUE before doing a GET or READ. From P$GET and P$RCH.

RT3009 - INPUT FORMAT ERROR ON INTEGER READ
The data to be read was not an integer. The type of the variable in the READ or READLN statement must match the type of the input data. From P$RI.

RT3010 - INPUT FORMAT ERROR ON REAL READ
The data to be read was not of type REAL. The type of the variable in the READ or READLN statement must match the type of the input data. From P$RF.

RT3011 - INPUT RECORD EXCEEDS BUFFER SIZE
If the program is reading a text file, the line length is too long. The maximum line length is 140 characters. If the program is reading a nontext file, the data is probably of the wrong type. Ensure that the data is of the same type as the variable specified to contain that data. From P$GET and P$RCH.

RT3012 - OUTPUT RECORD EXCEEDS BUFFER SIZE
The program tried to write more than 140 characters to a single line of a text file. Write an end-of-line (using the WRITELN statement) before reaching the 141st character. From P$PUT and P$WCH.

RT3013 - HARDWARE ERROR
A hardware error caused the program to fail. Attempt to run the program again. If this message continues to appear, contact a CRI site analyst. From any I/O routine.

RT3015 - ATTEMPTED EOLN ON NONTEXT FILE
The program invoked the EOLN function on a file other than a text file. A nontext file does not contain end-of-line indicators, and the EOLN function applies only to text files. Either remove the EOLN function or declare the file as type TEXT. From P$EOLN.

RT3016 - ATTEMPTED EOF ON UNSTRUCTURED FILE
The program invoked the EOF function on an unblocked dataset. From P$EOF.

RT3017 - CANNOT RESET OUTPUT OR $OUT
The RESET function was invoked on an output file. RESET is for input files only. Change RESET to REWRITE for an output file. From P$RESET.

RT3018 - CANNOT REWRITE INPUT OR $IN
The REWRITE function was invoked on an input file. REWRITE is for output files only. Change REWRITE to RESET for an input file. From P$REWRIT.

RT3019 - EOLN ATTEMPTED AT EOF
The EOLN standard function was called on a file currently positioned at the end of the file. EOLN is not defined when a file is at EOF. Change the program to prevent an EOLN test at EOF. From P$EOLN.

RT3020 - RESET OF UNDEFINED FILE
The program attempted to RESET an undefined file. Ensure that the file being RESET either exists before the program is run or has some data written to it before the RESET is done. From P$RESET.

E. PASCAL SYNTAX

This appendix defines all syntactical terms used in Cray Pascal, including those based on both the ISO standard and CRI extensions. Syntax is shown in the Backus-Naur Form (BNF).

Any word not enclosed in quotation marks is called a nonterminal symbol. Each nonterminal symbol in a BNF construction is defined in turn until the level of terminal (literal) symbols is reached. This progression of definitions from complex to simple is reversed in the index of syntax components (subsection E.2), which shows the uses of each component leading from simple to complex.

The following nonterminal symbols are defined but are not used to define other symbols:

compile-unit	simple-type	structured-type
pointer-type	special-symbol	signed-real
signed-number		

Directives, word-symbols, and identifiers disregard upper and lower case. Other BNF conventions are as follows:

Convention	Description
x | y	Indicates that either x or y is valid
"x"	Indicates that x is a literal, or terminal, element to be entered exactly as specified. This includes reserved words and standard identifiers, such as PROGRAM in the program heading. (Section 3, Pascal Vocabulary, lists the reserved words and standard identifiers.)
[x]	Indicates 0 or 1 occurrence of x is valid
{ x }	Indicates 0 or more occurrences of x are valid
Boldface	Indicates that the element is a CRI extension to the ISO Level 1 Pascal standard.
()	Indicates a grouping. Parentheses have no syntactic significance of their own. For example: id = (letter|"$"\|"%"\|"@") { letter\|digit\|"_"\|"$"|"%"|"@" }.
.	Indicates the end of a description

E.1 SYNTAX LISTING

This subsection defines nonterminal symbols using the Backus-Naur Form. Every component not enclosed by quotation marks is defined in turn until the level of terminal symbols is reached, indicated by quotation marks.

```
actual-parm =           expression |
                        var-access |
                        procedure-id |
                        function-id .

actual-parm-list =      "(" actual-parm  { "," actual-parm } ")" .

adding-operator =       "+" | "-" | "or" .

apostrophe-image =      "'" .

array-type =            "array" "[" index-type { "," index-type } "]"
                        "of" component-type .

array-value-spec =      [ type-id ]
                        "(" sub-value-spec { "," sub-value-spec } ")" .

array-var =             var-access .

assignment-statement =  ( var-access | function-id )
                        ":=" expression .

base-type =             ordinal-type .

block =                 {
                         label-dcl-part
                         constant-def-part
                         type-def-part
                         common-dcl-part
                         imported-dcl-part
                         exported-dcl-part
                         static-dcl-part
                         var-dcl-part
                        }
                        { procedure-and-function-dcl-part }
                        statement-part .

boolean-expression =    expression .

bound-id =              id .
```

```
buffer-var =            file-var "^" .

case-constant =         constant .

case-constant-list =    case-constant { "," case-constant } .

case-index =            expression .

case-list-element =     case-constant-list ":" statement .

case-statement =        "case" case-index "of"
                        case-list-element { ";" case-list-element }
                        ";" "otherwise" [ ":" ] statement
                        "end" .

character-string =      "'" string-element { string-element } "'" .

common-dcl-part =       "common" ext-var-dcl ";" { ext-var-dcl ";" } .

compile-unit =          ( program | module ) { program | module } .

component-type =        type-denoter .

component-var =         indexed-var | field-designator .

compound-statement =    "begin" statement-sequence "end" .

conditional-statement = if-statement | case-statement .

conformant-array-parm-spec =
                        value-conformant-array-spec |
                        var-conformant-array-spec .

conformant-array-schema =
                        packed-conformant-array-schema |
                        unpacked-conformant-array-schema .

constant =              ( [sign] (unsigned-number | constant-id ) ) |
                        character-string .

constant-def =          id "=" ( constant | constant-expression ) .

constant-def-part =     "const" constant-def ";" { constant-def ";" } .

constant-expression =   constant-subexpression |
                        ( "if" constant-subexpression
                        "then" constant-subexpression
                        "else" constant-subexpression ) .
```

```
constant-factor =       unsigned-constant |
                        constant-function-designator |
                        constant-set-constructor |
                        ( "(" constant-expression ")" ) |
                        ( "not" constant-factor ) .

constant-function-designator =
                        constant-function-id [actual-parm-list] .

constant-function-id =   id .

constant-id =           id .

constant-member-designator =
                        constant-expression [".." constant-expression ] .

constant-set-constructor =
                        "[" [ constant-member-designator
                        { "," constant-member-designator } ] "]" .

constant-subexpression =
                        simple-constant-expression
                        [ relational-operator
                        simple-constant-expression ] .

constant-term =         constant-factor
                        { multiplying-operator constant-factor } .

control-var =           entire-var .

digit =                 "0" | "1" | "2" | "3" | "4" |
                        "5" | "6" | "7" | "8" | "9" .

digit-sequence =        digit { digit } .

directive =             ( "forward" [ ";" "exported"
                        [ "(" external-name ")" ] ] ) |
                        ( "exported" [ "(" external-name ")" ]
                        [ ";" "forward" ] ) |
                        "fortran" |
                        "external" |
                        ( "imported" [ "(" external-name ")" ] ) .

directive-alt =         "exported" [ "(" external-name ")" ] .

domain-type =           type-id .

else-part =             "else" statement .

empty-statement =       .
```

```
entire-var =            var-id .

enumerated-type =       "(" id-list ")" .

exported-dcl-part =     "exported" ext-var-dcl ";" { ext-var-dcl ";" } .

expression =             ["if" subexpression "then" subexpression
                        "else"] subexpression.

ext-var-dcl =           id [ "(" external-name ")" ]
                        { "," id [ "(" external-name ")" ] }
                        ":" type-denoter .

extension-word-symbols =
                        "chache" | "common" | "imported" | "exported" |
                        "external" | "fortran | "imported" | "module" |
                        "otherwise" | "static" | "taskvar" | "value" |
                        "viewing" .

external-name =         id .

factor =                var-access |
                        unsigned-constant |
                        bound-id |
                        function-designator |
                        set-constructor |
                        ( "(" expression ")" ) |
                        ( "not" factor ) .

field-designator =      ( record-var "." field-specifier ) |
                        field-designator-id .

field-designator-id =   identifier .

field-list =            (
                        ( fixed-part [ ";" variant-part ] ) | variant-part
                        ) [ ";" ] .

field-specifier =       field-id .

file-type =             "file" "of" component-type .

file-var =              var-access .

final-value =           expression .

fixed-part =            record-section { ";" record-section } .

for-statement =         "for" control-var ":=" initial-value
                        ( "to" | "downto" ) final-value
                        ["by" increment-value]
                        "do" statement .
```

```
formal-parm-list =      "(" formal-parm-section
                        { ";" formal-parm-section } ")" .

formal-parm-section =   value-parm-spec |
                        var-parm-spec |
                        procedural-parm-spec |
                        functional-parm-spec |
                        conformant-array-parm-spec .

fractional-part =       digit-sequence .

function-block =        block .

function-dcl =          ( function-heading ";" directive ) |
                        ( function-identification ";" function-block ) |
                        ( function-heading ";"
                          [ directive-alt ";" ]
                          function-block ) .

function-designator =   function-id [ actual-parm-list ] .

function-heading =      "function" id [ formal-parm-list ]
                        ":" result-type .

function-id =           id .

function-identification = "function" function-id .

functional-parm-spec =  function-heading .

goto-statement =        "goto" label .

id =                    ( letter |"$"|"%"|"@" )
                        { letter | digit |"_"|"$"|"%"|"@" } .

id-list =               id { "," id } .

identified-var =        pointer-var "^" .

if-statement =          "if" boolean-expression "then" statement
                        [ else-part ] .

imported-dcl-part =     "imported" ext-var-dcl ";" { ext-var-dcl ";" } .

increment-value =       expression.

index-expression =      expression | ".." |
                        [ expression ".." expression ["..." expression ] ] .

index-type =            ordinal-type .
```

```
index-type-spec =       id ".." id ":" ordinal-type-id .

indexed-var =           array-var
                        "[" index-expression { "," index-expression } "]" .

initial-value =         expression .

label =                 digit-sequence .

label-dcl-part =        "label" label { "," label } ";" .

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".

member-designator =     expression [ ".." expression ] .

module =                "module" id ";" module-block "." .

module-block =          {
                         constant-def-part
                         type-def-part
                         common-dcl-part
                         imported-dcl-part
                         exported-dcl-part
                         static-dcl-part
                         var-dcl-part
                        }
                        [ value-def-part ]
                        { procedure-and-function-dcl-part } .

multiplying-operator =  "*" | "/" | "div" | "mod" | "and" .

new-ordinal-type =      enumerated-type | subrange-type .

new-pointer-type =      "^" domain-type .

new-structured-type =   [ "packed" ] unpacked-structured-type .

new-type =              new-ordinal-type |
                        new-structured-type |
                        new-pointer-type .

octal-digit =           '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7' .

octal-number =          octal-digit { octal-digit } 'B' .

ordinal-type =          new-ordinal-type | ordinal-type-id .

ordinal-type-id =       type-id .
```

```
packed-conformant-array-schema =
                        "packed" "array" "[" index-type-spec "]"
                        "of" type-id .

pointer-type =          new-pointer-type | pointer-type-id .

pointer-type-id =       type-id .

pointer-var =           var-access .

procedural-parm-spec =  procedure-heading .

procedure-and-function-dcl-part =
                        { ( procedure-dcl | function-dcl ) ";" } .

procedure-block =       block .

procedure-dcl =         ( procedure-heading ";" directive ) |
                        ( procedure-identification ";" procedure-block ) |
                        ( procedure-heading ";"
                          [ directive-alt  ";" ]
                          procedure-block ) .

procedure-heading =     "procedure" id [ formal-parm-list ] .

procedure-id =          id .

procedure-identification = "procedure" id .

procedure-statement =   procedure-id
                        [ [ actual-parm-list ] |
                        read-parm-list |
                        readln-parm-list |
                        write-parm-list |
                        writeln-parm-list ] .

program =               program-heading ";" program-block "." .

program-block =         {
                         label-dcl-part
                         constant-def-part
                         type-def-part
                         common-dcl-part
                         imported-dcl-part
                         exported-dcl-part
                         static-dcl-part
                         var-dcl-part
                        }
                        [value-def-part]
                        { procedure-and-function-dcl-part }
                        statement-part .
```

```
program-heading =       "program" id [ "(" program-parms ")" ] .

program-parms =         id-list .

read-parm-list =        "(" [ file-var "," ]
                        var-access { "," var-access } ")" .

readln-parm-list =      "(" ( file-var | var-access )
                        { "," var-access } ")" .

real-type-id =          type-id .

record-section =        record-section-id-list ":" type-denoter .

record-section-id =     id .

record-section-id-list = record-section-id { "," record-section-id } .

record-type =           "record" field-list "end" .

record-value-spec =     [ type-id ]
                        "(" value-spec { "," value-spec } ")" .

record-var =            var-access .

record-var-list =       record-var { "," record-var } .

relational-operator =   "=" | "<>" | "<" | ">" | "<=" | ">=" | "in" .

repeat-statement =      "repeat" statement-sequence
                        "until" boolean-expression .

repetitive-statement =  repeat-statement |
                        while-statement |
                        for-statement .

result-type =           simple-type-id | pointer-type-id .

scale-factor =          signed-integer .

set-constructor =       "[" [ member-designator
                        { "," member-designator } ] "]" .

set-id =                id .

set-type =              "set" "of" base-type .

set-value-elt =         constant [ ".." constant ] .

set-value-elt-list =    set-value-elt { "," set-value-elt } .
```

```
set-value-spec =        set-id | "[" [ set-value-elt-list ] "]" .

sign =                  "+" | "-" .

signed-integer =        [ sign ] unsigned-integer .

signed-number =         signed-integer |
                        signed-real |
                        ( [ sign ] octal-number ) .

signed-real =           [ sign ] unsigned-real .

simple-constant-expression =
                        [ sign ] constant-term
                        { adding-operator constant-term } .

simple-expression =     [ sign ] term { adding-operator term } .

simple-statement =      empty-statement |
                        assignment-statement |
                        procedure-statement |
                        goto-statement .

simple-type =           ordinal-type | real-type-id .

simple-type-id =        type-id .

simple-value-spec =     unsigned-constant |
                        sign unsigned-number |
                        sign constant-id.

special-symbol =        "+" | "-" | "*" | "/" | "=" | "<" | ">" | "[" |
                        "]" | "." | "," | ":" | ";" | "^" | "(" | ")" |
                        "<>" | "<=" | ">=" | ":=" | ".." | word-symbol .

statement =             [ label ":" ]
                        ( simple-statement | structured-statement ) .

statement-part =        compound-statement .

statement-sequence =    statement { ";" statement } .

static-dcl-part =       [ "static" var-dcl ";" { var-dcl ";" } ] .

string-character =      one-of-a-set-of-the-printable-ascii-characters .

string-element =        apostrophe-image | string-character .
```

```
structured-statement =  compound-statement |
                        conditional-statement |
                        repetitive-statement |
                        with-statement |
                        viewing-statement .

structured-type =       new-structured-type | structured-type-id .

structured-type-id =    type-id .

subexpression =         simple-expression
                        [ relational-operator simple-expression ].

subrange-type =         constant ".." constant .

sub-value-spec =        value-spec |
                        ( unsigned-integer | constant-id )
                        "of" value-spec .

tag-field =             identifier .

tag-type =              ordinal-type-id .

taskvar-var-dcl-part =  [ "taskvar" ext-var-decl ";"
                        { ext-var-decl ";" } ] .

term =                  factor { multiplying-operator factor } .

type-def =              id "=" type-denoter .

type-def-part =         "type" type-def ";" { type-def ";" } .

type-denoter =          type-id | new-type .

type-id =               id .

unpacked-conformant-array-schema =
                        "array"
                        "[" index-type-spec { ";" index-type-spec } "]"
                        "of" ( type-id | conformant-array-schema ) .

unpacked-structured-type =
                        array-type | record-type | set-type | file-type .

unsigned-constant =     unsigned-number |
                        character-string |
                        constant-id |
                        "nil" .

unsigned-integer =      digit-sequence .
```

```
unsigned-number =       unsigned-integer | unsigned-real | octal-number .

unsigned-real =         ( unsigned-integer "."
                          fractional-part
                          [ "e" scale-factor ] ) |
                        ( unsigned-integer "e" scale-factor ) .

value-conformant-array-spec = id-list ":" conformant-array-schema .

value-def =             entire-var "=" value-spec .

value-def-part =        "value" value-def ";" { value-def ";" } .

value-parm-spec =       id-list ":" type-id .

value-spec =            simple-value-spec |
                        set-value-spec |
                        array-value-spec |
                        record-value-spec .

var-access =            entire-var |
                        component-var |
                        identified-var |
                        buffer-var .

var-conformant-array-spec = "var" id-list ":" conformant-array-schema .

var-dcl =               id-list ":" type-denoter .

var-dcl-part =          "var" var-dcl ";" { var-dcl ";" } .

var-id =                id .

var-parm-spec =         "var" id-list ":" type-id .

variant =               case-constant-list ":" "(" field-list ")" .

variant-part =          "case" variant-selector "of" variant
                        { ";" variant } .

variant-selector =      [ tag-field ":" ] tag-type .

viewing-statement =     "viewing" var-access ":" type-id "do" statement .

while-statement =       "while" boolean-expression "do" statement .

with-statement =        "with" record-var-list "do" statement .
```

```
word-symbol =          "and"      | "array"    | "begin"    | "by"         |
                       "case"     | "const"    | "div"      | "do"         |
                       "downto"   | "else"     | "end"      | "file"       |
                       "for"      | "fortran"  | "forward"  | "function"   |
                       "goto"     | "if"       | "in"       | "label"      |
                       "mod"      | "nil"      | "not"      | "of"         |
                       "or"       | "packed"   | "procedure"| "program"    |
                       "record"   | "repeat"   | "set"      | "then"       |
                       "to"       | "type"     | "until"    | "var"        |
                       "while"    | "with"     |
                       extension-word-symbols .

write-parameter =      expression [ ":" expression [ ":" expression ] ] .

write-parm-list =      "(" [ file-var "," ]
                       write-parm { "," write-parm } ")" .

writeln-parm-list =    [ "(" ( file-var | write-parm )
                       { "," write-parm } ")" ] .
```

E.2 INDEX OF SYNTAX COMPONENTS

This subsection shows which BNF constructions use a given syntactical component. This allows a study of the applications of a given concept in Pascal. Literal symbols, indicated by quotation marks, are listed at the end of the index.

```
"case" : case-statement, variant-part, word-symbol
case-constant : case-constant-list
case-constant-list : case-list-element, variant
case-index : case-statement
case-list-element : case-statement
case-statement : conditional-statement
character-string : constant, unsigned-constant
"common" : common-dcl-part, extension-word-symbols
common-dcl-part : block, module-block
component-type : array-type, file-type
component-var : var-access
compound-statement : statement-part, structured-statement
conditional-statement : structured-statement
conformant-array-parm-spec : formal-parm-section
conformant-array-schema : unpacked-conformant-array-schema, value-
        conformant-array-spec, var-conformant-array-spec
"const" : constant-def-part, word-symbol
constant : case-constant, constant-def, set-value-elt, subrange-type
constant-def : constant-def-part
constant-def-part : block, program-block, module-block
constant-expression : constant-def, constant-factor,
        constant-member-designator
constant-factor : constant-term
constant-function-designator : constant-factor
constant-function-id : constant-function-designator
constant-id : constant, unsigned-constant, simple-value-spec,
        sub-value-spec, unsigned-constant

constant-set-constructor : constant-factor
constant-subexpression : constant-expression
constant-term : simple-constant-expression
control-var : for-statement

digit : digit-sequence, id
digit-sequence : fractional-part, label, unsigned-integer
directive : function-dcl, procedure-dcl
directive-alt : function-dcl, procedure-dcl
"div" : multiplying-operator, word-symbol
"do" : for-statement, viewing-statement, while-statement, with-statement,
        word-symbol
domain-type : new-pointer-type
"downto" : for-statement, word-symbol

"else" : constant-expression, else-part, expression, word-symbol
else-part : if-statement
empty-statement : simple-statement
"end" : case-statement, compound-statement, record-type, word-symbol
entire-var : control-var, value-def, var-access
enumerated-type : new-ordinal-type
"exported" : directive-alt, directive, exported-dcl-part, extension-word-
        symbols
```

```
exported-dcl-part : block, program-block, module-block
expression : actual-parm, assignment-statement, boolean-expression, case-
         index, factor, final-value, increment-value, index-expression,
         initial-value, member-designator, member-designator,
         write-parameter
ext-var-dcl : common-dcl-part, exported-dcl-part, imported-dcl-part,
         taskvar-var-dcl-part
extension-word-symbols : word-symbol
"external" : directive
external-name : directive, directive-alt, ext-var-dcl

factor : factor, term
field-designator : component-var
field-designator-id : field-designator
field-id : field-specifier
field-list : record-type, variant
field-specifier : field-designator
"file" : file-type, word-symbol
file-type : unpacked-structured-type
file-var : buffer-var, read-parm-list, readln-parm-list, write-parm-list,
         writeln-parm-list
final-value : for-statement
fixed-part : field-list
"for" : for-statement, word-symbol
for-statement : repetitive-statement
formal-parm-list : function-heading, procedure-heading
formal-parm-section : formal-parm-list
"fortran" : directive
"forward" : directive
fractional-part : unsigned-real
"function" : function-heading, function-identification, word-symbol
function-block : function-dcl
function-dcl : procedure-and-function-dcl-part
function-designator : factor
function-heading : function-dcl, functional-parm-spec
function-id : actual-parm, assignment-statement, function-designator,
         function-identification
function-identification : function-dcl
functional-parm-spec : formal-parm-section

"goto" : goto-statement, word-symbol
goto-statement : simple-statement

id or identifier : bound-id, constant-def, constant-function-id,
         constant-id, ext-var-dcl, external-name, field-identifier-id,
         function-heading, function- id, id-list, index-type-spec,
         module, procedure-heading, procedure-id,
         procedure-identification, program-heading, record-section-id,
         set-id, tag-field, type-def, type-id, var-id
identified-var : var-access
```

```
id-list : enumerated-type, program-parms, value-conformant-array-spec,
          value-parm-spec, var-conformant-array-spec, var-dcl,
          var-parm-spec
"if" : constant-expression, expression, if-statement, word-symbol
if-statement : conditional-statement
"imported" : directive, extension-word-symbols, imported-dcl-part
imported-dcl-part : block, program-block, module-block
"in" : relational-operator, word-symbol
increment-value : for-statement
index-expression : indexed-var
index-type : array-type
index-type-spec : packed-conformant-array-schema, unpacked-conformant-
          array-schema
indexed-var : component-var
initial-value : for-statement

"label" : label-dcl-part, word-symbol
label : goto-statement, label-dcl-part, statement
label-dcl-part : block, program-block
letter : id

member-designator : set-constructor
"mod" : multiplying-operator, word-symbol
"module" : extension-word-symbols, module
module : compile-unit
module-block : module
multiplying-operator : constant term, term

new-ordinal-type : new-type, ordinal-type
new-pointer-type : new-type, pointer-type
new-structured-type : new-type, structured-type
new-type : type-denoter
"nil" : unsigned-constant, word-symbol
"not" : constant-factor, factor, word-symbol

octal-digit : octal-number
octal-number : unsigned-number, signed-number
"of" : array-type, case-statement, file-type, packed-conformant-array-
          schema, set-type, sub-value-spec, unpacked-conformant-array-
          schema, variant-part, word-symbol
one-of-a-set-of-the-printable-ascii-characters : string-character
"or" : adding-operator, word-symbol
ordinal-type : base-type, index-type, simple-type
ordinal-type-id : index-type-spec, ordinal-type, tag-type
"otherwise" : extension-word-symbols, case-statement

"packed" : new-structured-type, packed-conformant-array-schema,
          word-symbol
packed-conformant-array-schema : conformant-array-schema
pointer-type-id : pointer-type, result-type
pointer-var : identified-var
```

```
procedural-parm-spec : formal-parm-section
"procedure" : procedure-heading, procedure-identification, word-symbol
procedure-and-function-dcl-part : block, program-block, module-block
procedure-block : procedure-dcl
procedure-dcl : procedure-and-function-dcl-part
procedure-heading : procedural-parm-spec, procedure-dcl
procedure-id : actual-parm, procedure-statement
procedure-identification : procedure-dcl
procedure-statement : simple-statement
"program" : program-heading, word-symbol
program : compile-unit
program-block : program
program-heading : program
program-parms : program-heading

read-parm-list : procedure-statement
readln-parm-list : procedure-statement
real-type-id : simple-type
"record" : record-type, word-symbol
record-section : fixed-part
record-section-id : record-section-id-list
record-section-id-list : record-section
record-type : unpacked-structured-type
record-value-spec : value-spec
record-var : field-designator, record-var-list
record-var-list : with-statement
relational-operator : constant-subexpression, subexpression
"repeat" : repeat-statement, word-symbol
repeat-statement : repetitive-statement
repetitive-statement : structured-statement
result-type : function-heading

scale-factor : unsigned-real
"set" : set-type, word-symbol
set-constructor : factor
set-id : set-value-spec
set-type : unpacked-structured-type
set-value-elt : set-value-elt-list
set-value-elt-list : set-value-spec
set-value-spec : value-spec
sign : constant, signed-integer, signed-real, simple-constant-expression,
         simple-expression, simple-value-spec
signed-integer : scale-factor, signed-number
simple-constant-expression : constant-subexpression
simple-expression : subexpression
simple-statement : statement
simple-type-id : result-type
simple-value-spec : value-spec
statement : case-list-element, case-statement, else-part, for-statement,
         if-statement, statement-sequence, viewing-statement, while-
         statement, with-statement
```

```
statement-part : block, program-block
statement-sequence : compound-statement, repeat-statement
"static" : extension-word-symbols, static-dcl-part
static-dcl-part : block, program-block, module-block
string-character : string-element
string-element : character-string
structured-statement : statement
structured-type-id : structured-type
subexpression : expression
sub-value-spec : array-value-spec
subrange-type : new-ordinal-type

tag-field : variant-selector
tag-type : variant-selector
"taskvar" : taskvar-var-dcl-part
term : simple-expression
"then" : constant-expression, expression, if-statement, word-symbol
"to" : for-statement, word-symbol
"type" : type-def-part, word-symbol
type-def : type-def-part
type-def-part : block, program-block, module-block
type-denoter : component-type, ext-var-dcl, record-section, type-def,
        var-dcl
type-id : array-value-spec, domain-type, ordinal-type-id, packed-
        conformant-array-schema, pointer-type-id, real-type-id,
        record-value-spec, simple-type-id, structured-type-id, type-
        denoter, unpacked-conformant-array-schema, value-parm-spec,
        var-parm-spec, viewing-statement
unpacked-conformant-array-schema : conformant-array-schema
unpacked-structured-type : new-structured-type
unsigned-constant : constant-factor, factor, simple-value-spec
unsigned-integer : signed-integer, sub-value-spec, unsigned-number,
        unsigned-real
unsigned-number : constant, simple-value-spec, unsigned-constant
unsigned-real : signed-real, unsigned-number
"until" : repeat-statement, word-symbol

"value" : extension-word-symbols, value-def-part
value-conformant-array-spec : conformant-array-parm-spec
value-def : value-def-part
value-def-part : module-block, program-block
value-parm-spec : formal-parm-section
value-spec : record-value-spec, sub-value-spec, value-def
"var" : var-conformant-array-spec, var-dcl-part, var-parm-spec,
        word-symbol
var-access : actual-parm, array-var, assignment-statement, factor,
        file-var, pointer-var, read-parm-list, readln-parm-list,
        record-var, viewing-statement
var-conformant-array-spec : conformant-array-parm-spec
var-dcl : static-dcl-part, var-dcl-part
var-dcl-part : block, program-block, module-block
```

```
var-id : entire-var
var-parm-spec : formal-parm-section
variant : variant-part
variant-part : field-list
variant-selector : variant-part
"viewing" : viewing-statement
viewing-statement : structured-statement

"while" : while-statement, word-symbol
while-statement : repetitive-statement
"with" : with-statement, word-symbol
with-statement : structured-statement
word-symbol : special-symbol
write-parm : write-parm-list, writeln-parm-list
write-parm-list : procedure-statement
writeln-parm-list : procedure-statement

"a" through "z" : letter

"0" through "7" : digit, octal-digit
"8" and "9" : digit

"_" : id
"^" : buffer-var, identified-var, new-pointer-type, special-symbol
"<" : relational-operator, special-symbol
"<|" : relational-operator, special-symbol
"|" : relational-operator, special-symbol
"|=" : relational-operator, special-symbol
"<=" : relational-operator, special-symbol
"$" : id
"%" : id
"'" : character-string
"''" : apostrophe-image
")" and "(" : actual-parm-list, array-value-spec, directive-alt,
        directive, enumerated-type, ext-var-dcl, factor,
        formal-parm-list, program-heading, read-parm-list,
        readln-parm-list, record-value-spec, special-symbol, variant,
        write-parm-list, writeln-parm-list
"*" : multiplying-operator, special-symbol
"+" : adding-operator, sign, special-symbol
"," : actual-parm-list, array-type, array-value-spec, case-constant-list,
        ext-var-dcl, id-list, indexed-var, label-dcl-part,
        read-parm-list, readln-parm-list, record-section-id-list,
        record-value-spec, record-var-list, set-constructor, set-value-
        elt-list, special-symbol, write-parm-list, writeln-parm-list
"-" : adding-operator, sign, special-symbol
"." : field-designator, module, program, special-symbol, unsigned-real
".." : index-type-spec, member-designator, set-value-elt, special-symbol,
        subrange-type
"/" : multiplying-operator, special-symbol
```

":" : case-list-element, case-statement, ext-var-dcl, function-heading, index-type-spec, record-section, special-symbol, statement, value-conformant-array-spec, value-parm-spec, var-conformant-array-spec, var-dcl, var-parm-spec, variant-selector, variant, write-parameter

":=" : assignment-statement, for-statement, special-symbol

";" : case-statement, common-dcl-part, constant-def-part, directive, exported-dcl-part, field-list, fixed-part, formal-parm-list, function-dcl, imported-dcl-part, label-dcl-part, module, procedure-and-function-dcl-part, procedure-dcl, program, special-symbol, statement-sequence, static-dcl-part, taskvar-var-dcl-part, type-def- part, unpacked-conformant-array-schema, value-def-part, value-dcl-part, variant-part

"=" : constant-def, relational-operator, special-symbol, type-def, value-def

"@" : id

"]" and "[" : array-type, indexed-var, packed-conformant-array-schema, set-constructor, set-value-spec, special-symbol, unpacked-conformant-array-schema

F. DEBUG INFORMATION

Compiler options D+, DM*n*, and BP+ provide debugging information to help locate problems in a Pascal source program. The D+ option generates a *walkback* through the program when a fatal error or a HALT is encountered. The DM*n* option, with *n*>1, allows DEBUG to write a formatted dump of the program. The BP+ option provides for a data breakpoint on a specified memory write. (See section 2, Using Pascal on a Cray Computer, for descriptions of compiler options and directives.)

For more debugging information, see the Symbolic Debugging Package Reference Manual, CRI publication SR-0112.

F.1 D+ DEBUGGING INFORMATION

When a run-time error or a call to the predefined procedure HALT is encountered during the execution of a Pascal program, a library routine takes control and performs the following functions:

- Prints the approximate line in the source program where the error or call to HALT occurred
- Produces a stack *walkback*

The walkback is a listing of active procedures and functions, with the most recently called listed first, and the approximate line in the source program from which each was called.

If the program is compiled with the D+ option in effect, the unstructured variables for each active procedure and function are also listed. Due to the recursive possibilities of Pascal, a procedure or function may be activated many times in the course of a program. Only the unstructured variables for the three most recent activations are dumped, however.

The following program is followed by the debugging information available through this facility:

```
    1  program debug_example (output);
    2
    3          var
    4               i, j: integer;
    5               x, y: Boolean;
    6               a, b: (cat, mouse, frog, fruit_bat);
    7               r, s: real;
    8               t, u: * integer;
    9               qwert: char;
           10
           11     procedure bozhemoi (i: integer);
           12        var
           13           b: integer;
           14           a: Boolean;
           15           c: char;
           16           bx: Boolean;
           17           Trondheim_Hammer_Dance: real;
           18     begin
     1     19        if i < 7 then
     2     20           bozhemoi (i + 1)
     1     21        else
     2     22           halt
           23        end;
           24
           25     begin (*debug_example*)
           26
     1     27      i := 1;
     1     28      j := 2;
     1     29      x := true;
     1     30      y := false;
     1     31      a := mouse;
     1     32      b := frog;
     1     33      r := 3.02;
     1     34      s := -125;
     1     35      new (t);
     1     36      u := nil;
     1     37      qwert := 'X';
     1     38      bozh0moi (1)
     1     39
           40      end (*debug_example*).
```

**** **** No errors **** ****

```
**** **** COMMENT - B04..B14 reserved for parameter passing.
**** **** COMMENT - B15..B60 reserved for local variables of nested
                     procedures and functions.
**** **** COMMENT - B61 assigned to global variable A at @MAIN+000015
**** **** COMMENT - B62 assigned to global variable B at @MAIN+000016
**** **** COMMENT - B63 assigned to global variable T at @MAIN+000017
**** **** COMMENT - B64 assigned to global variable U at @MAIN+000020
**** **** COMMENT - B65 assigned to global variable QWERT at @MAIN+000021
**** **** COMMENT - T00..T07 reserved for parameter passing.
**** **** COMMENT - T10..T61 reserved for local variables of nested
                     procedures and functions.
**** **** COMMENT - T62 assigned to global variable I at @MAIN+000022
**** **** COMMENT - T63 assigned to global variable J at @MAIN+000023
**** **** COMMENT - T64 assigned to global variable X at @MAIN+000024
**** **** COMMENT - T65 assigned to global variable Y at @MAIN+000025
**** **** COMMENT - T66 assigned to global variable R at @MAIN+000026
**** **** COMMENT - T67 assigned to global variable S at @MAIN+000027

Procedure and function list, with declaration line numbers

     1   DEBUG_EXAMPLE (program)
    11   : BOZHEMOI External and common block names
       $END     : entry, imported
       $STKOFEN : entry, imported
       DEBUG_EX : start, exported at 000410a relative
       P$HALT   : entry, imported
       P$NEW    : entry, imported
       P$RUNTIM : entry, imported
**** **** END PASCAL : 40 lines, 0 errors.

END:heap size                      = 170388 words
END:allocated areas                = 2
Initial runtime stack requested    = 4K words
Runtime stack increment requested  = 4K words
Initial runtime heap requested     = 2K words
Runtime heap increment requested   = 20K words
/EOF


*** Runtime error RT1000: program called HALT Stack walkback begun in
      P$HALT at 00001176d (line 41)
```

```
P$HALT was called from BOZHEMOI at 00000554d (line 23)
BOZHEMOI was called from BOZHEMOI at 00000545a (line 20)
    A         FALSE
    B                              0
    BX        FALSE
    C         ' '
    I                              7
    TRONDHE    0.0000000000000E+0000
BOZHEMOI was called from BOZHEMOI at 00000545a (line 20)
    A         FALSE
    B                              0
    BX        FALSE
    C         ' '
    I                              6
    TRONDHE    0.0000000000000E+0000
BOZHEMOI was called from BOZHEMOI at 00000545a (line 20)
    A         FALSE
    B                              0
    BX        FALSE
    C         ' '
    I                              5
    TRONDHE    0.0000000000000E+0000
BOZHEMOI was called from BOZHEMOI at 00000545a (line 20)
    A         FALSE
    B                              0
    BX        FALSE
    C         ' '
    I                              4
    TRONDHE    0.0000000000000E+0000
BOZHEMOI was called from BOZHEMOI at 00000545a (line 20)
  Only last 3 activations dumped.
BOZHEMOI was called from BOZHEMOI at 00000545a (line 20)
  Only last 3 activations dumped.
BOZHEMOI was called from DEBUG_EX at 00000651d (line 38)
  Only last 3 activations dumped.
DEBUG_EX
    A                              1
    B                              2
    I                              1
    J                              2
    QWERT     'X'
    R              3.0200000000000E+0000
    S             -1.2500000000000E+0002
    T            00067177
    U            00000000
    X            TRUE
    Y            FALSE
***End walkback.
*** Error RT1000 is fatal.
```

F.2 BP+ DEBUGGING INFORMATION

Enabling the BP+ compiler option generates data breakpoints on specified memory writes. If it is used to trace a variable, this option writes a line of output each time a value is assigned to the variable.

To use this facility, the following procedure declaration must appear in the declarations section of the main program:

```
PROCEDURE P$BREAK (VAR location: INTEGER; isvariable: BOOLEAN);
                  IMPORTED;
```

To watch the static variable k, the following must appear before the first occurrence of k to be watched:

```
P$BREAK (k, TRUE )
```

Alternatively, the following could be specified to watch the absolute address 1027:

```
K := 1027;
P$BREAK(k,FALSE);
```

A program containing both the P$BREAK declaration and one of the preceding calls to P$BREAK is then recompiled with the BP+ option set. A second recompile may be necessary when calling P$BREAK with isvariable=FALSE, since the word being watched may move when breakpoint code is inserted.

The BP+ option generates the following CAL code in front of every line of Pascal code:

```
A0      linenumber
R       P$DBP
```

Procedure P$DBP checks the value of the location defined by the call to P$BREAK and reports when it changes. P$DBP restores all registers except B00 before returning.

This facility increases the size of compiled code by 1 word per user statement and decreases the speed of execution by as much as 50 percent.

The following sample program writes the value of the variable i whenever a new value is assigned to it:

```
CRAY PASCAL (03.00) 16:28:04 08/08/85                                PAGE 1
    1  PROGRAM test105(OUTPUT);
    2  VAR i,j,k : INTEGER;
    3  PROCEDURE P$BREAK ( VAR i : INTEGER; isvar : BOOLEAN); IMPORTED;
    4  BEGIN
    5    P$BREAK(i,TRUE);
    6    WRITELN(' RUNNING...');
    7    FOR i := 1 TO 20 DO
    8      BEGIN
    9        j := 1;
   10      END;
   11    i := 1;
   12    WHILE i < 10 DO
   13      i := i + 1;
   14    REPEAT
   15      i := i - 1;
   16    UNTIL i = 0;
   17    IF i = 0
   18    THEN i := 5
   19    ELSE i := 6;
   20    WRITELN(' ALL FINISHED, SIR.');
   21  END.

**** **** NO ERRORS **** ****
EXTERNAL AND COMMON BLOCK NAMES
       P$BREAK  : ENTRY, IMPORTED
       P$DBP    : ENTRY, IMPORTED
       P$PUT    : ENTRY, IMPORTED
       P$RUNTIM : ENTRY, IMPORTED
       P$WSTR   : ENTRY, IMPORTED
       test105  : START, EXPORTED AT 000311A RELATIVE
**** **** END PASCAL : 21 LINES, 0 ERRORS.
RUNTIME STACK REQUESTED = 20K WORDS
RUNTIME HEAP REQUESTED  = 2K WORDS
          RELOCATABLE LOAD
LOAD TRANSFER IS TO    TEST105   AT  (     511a )
 DATASET    BLOCK       ADDRESS   LENGTH     DATE    OS REV    PROCSSR
VER.       COMMENT
           *SYSTEM       0       200
$BLD       TEST105     200       410  08/08/83  COS 1.12  PASCAL     (01.00)
$PSCLIB    P$DBP       610      1517  08/03/83  COS 1.12  CAL X.12  04/22/83
           P$CALLR    2327        22  08/03/83  COS 1.12  CAL X.12  04/22/83
           P$DEBUG    2351      1547  08/03/83  COS 1.12  CAL X.12  04/22/83
           P$GET      4120       170  08/03/83  COS 1.12  CAL X.12  04/22/83
           P$$$HPAD   4310        12  08/03/83  COS 1.12  CAL X.12  04/22/83
           P$RUNTIM   4322      1101  08/03/83  COS 1.12  CAL X.12  04/22/83
           P$OPEN     5423       207  08/03/83  COS 1.12  CAL X.12  04/22/83
```

	P$PUT	5632	130	08/03/83	COS 1.12	CAL X.12	04/22/83
	P$RESET	5762	100	08/03/83	COS 1.12	CAL X.12	04/22/83
	P$TRACE	6062	2252	08/03/83	COS 1.12	CAL X.12	04/22/83
	P$WCH	10334	27	08/03/83	COS 1.12	CAL X.12	04/22/83
	P$WI	10363	113	08/03/83	COS 1.12	CAL X.12	04/22/83
	P$WSTR	10476	63	08/03/83	COS 1.12	CAL X.12	04/22/83
	P$MEMRY	10561	3062	08/03/83	COS 1.12	PASCAL	(01.00)
	P$NEW	13643	306	08/03/83	COS 1.12	PASCAL	(01.00)
	P$EOF	14151	137	08/03/83	COS 1.12	PASCAL	(01.00)
$SYSLIB	$BKSP	14310	113	05/27/83	COS X.12	CAL X.12	04/22/83
	$DSNDSP	14423	26	05/27/83	COS X.12	CAL X.12	04/22/83
	GPOS	14451	121	05/27/83	COS X.12	CAL X.12	04/22/83
	$GTDSP	14572	116	05/27/83	COS X.12	CAL X.12	04/22/83
	$INSASCI	14710	70	05/27/83	COS X.12	CAL X.12	04/22/83
	$MEMORY	15000	324	05/27/83	COS X.12	CAL X.12	04/22/83
	$MEMAUTO	15324	33	05/27/83	COS X.12	CAL X.12	04/22/83
	$MEMFLMX	15357	54	05/27/83	COS X.12	CAL X.12	04/22/83
	$MEMUC20	15433	276	05/27/83	COS X.12	CAL X.12	04/22/83
	PACK	15731	47	05/27/83	COS X.12	CAL X.12	04/22/83
	$PBN	16000	157	05/27/83	COS X.12	CAL X.12	04/22/83
	$PDD	16157	4	05/27/83	COS X.12	CAL X.12	04/22/83 PDD TABLE
	$PRCW	16163	120	05/27/83	COS X.12	CAL X.12	04/22/83
	$RCW	16320	651	05/27/83	COS X.12	CAL X.12	04/22/83
	$SDSP	17171	46	05/27/83	COS X.12	CAL X.12	04/22/83
	$SLERP	17237	1062	05/27/83	COS X.12	CAL X.12	04/22/83
	SPOS	20321	362	05/27/83	COS X.12	CAL X.12	04/22/83
	$TRBK	20703	303	05/27/83	COS X.12	CAL X.12	04/22/83
	TRBKLVL%	21206	57	05/27/83	COS X.12	CAL X.12	04/22/83
	$UEOFTCL	21265	13	05/27/83	COS X.12	CAL X.12	04/22/83
	$WCW	21300	1144	05/27/83	COS X.12	CAL X.12	04/22/83
$FTLIB	$BTD	22460	37	06/21/83	COS X.12	CAL X.12	04/22/83
	$BTO	22520	35	06/21/83	COS X.12	CAL X.12	04/22/83
	$DASS	22560	74	06/21/83	COS X.12	CAL X.12	04/22/83
	$DDSS	22660	43	06/21/83	COS X.12	CAL X.12	04/22/83
	$DMSS	22740	63	06/21/83	COS X.12	CAL X.12	04/22/83
	$GETPOS	23023	33	06/21/83	COS X.12	CAL X.12	04/22/83
	$IBMPACK	23056	146	06/21/83	COS X.12	CAL X.12	04/22/83
	$IBMTRAN	23224	1032	06/21/83	COS X.12	CAL X.12	04/22/83
	$IOERP	24256	1537	06/21/83	COS X.12	CAL X.12	04/22/83
	$IUO	26015	416	06/21/83	COS X.12	CAL X.12	04/22/83
	$NCON	26433	173	06/21/83	COS X.12	CAL X.12	04/22/83
	$NOCV	26626	452	06/21/83	COS X.12	CAL X.12	04/22/83
	$READ	27300	33	06/21/83	COS X.12	CAL X.12	04/22/83
	$SETPOS	27333	35	06/21/83	COS X.12	CAL X.12	04/22/83
	$UTIL	27370	164	06/21/83	COS X.12	CAL X.12	04/22/83
	$WFD	27554	2046	06/21/83	COS X.12	CAL X.12	04/22/83
	$WUT	31622	1052	06/21/83	COS X.12	CAL X.12	04/22/83

```
*** LOAD IMAGE STATISTICS ***
ABSOLUTE BINARY LENGTH:   13756(10),      32674(8) WORDS
PROGRAM IMAGE: FWA =        200(8),    LWA =      33074(8)

DATA B.P.:ADDR=00000000, CONTENTS= 0000000003267400000000
     BETWEEN LINES        1 AND        9
P$BREAK  :ADDR=00000215, CONTENTS= 00000000000000000000
RUNNING...
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000001
     BETWEEN LINES       11 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000002
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000003
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000004
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000005
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000006
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000007
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000010
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000011
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000012
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000013
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000014
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000015
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000016
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000017
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000020
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000021
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000022
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000023
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000024
     BETWEEN LINES       13 AND       13
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000025
     BETWEEN LINES       13 AND       15
```

```
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000001
     BETWEEN LINES       15 AND       16
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000002
     BETWEEN LINES       17 AND       17
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000003
     BETWEEN LINES       17 AND       17
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000004
     BETWEEN LINES       17 AND       17
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000005
     BETWEEN LINES       17 AND       17
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000006
     BETWEEN LINES       17 AND       17
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000007
     BETWEEN LINES       17 AND       17
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000010
     BETWEEN LINES       17 AND       17
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000011
     BETWEEN LINES       17 AND       17
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000012
     BETWEEN LINES       17 AND       18
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000011
     BETWEEN LINES       19 AND       19
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000010
     BETWEEN LINES       19 AND       19
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000007
     BETWEEN LINES       19 AND       19
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000006
     BETWEEN LINES       19 AND       19
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000005
     BETWEEN LINES       19 AND       19
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000004
     BETWEEN LINES       19 AND       19
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000003
     BETWEEN LINES       19 AND       19
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000002
     BETWEEN LINES       19 AND       19
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000001
     BETWEEN LINES       19 AND       19
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000000
     BETWEEN LINES       19 AND       21
DATA B.P.:ADDR=00000215, CONTENTS= 00000000000000000005
     BETWEEN LINES       22 AND       25
ALL FINISHED, SIR.
```

G. ERRORS NOT REPORTED BY CRAY PASCAL

Errors listed here are specified in the ISO Level 1 Pascal standard but are not reported by Cray Pascal. The error designation from appendix D, Run-time Messages, of the standard follows each error in parentheses:

- It is an error unless a variant is active for the entirety of each reference and access to each component of the variant (D.2).
- It is an error to alter the value of a file-variable f when a reference to the buffer variable f^ exists (D.6).
- It is an error if the buffer-variable is undefined immediately prior to any use of PUT (D.12).
- For NEW(p,c1,...,cn), it is an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants (D.19).
- For DISPOSE(p), it is an error if the identifying-value had been created using the form NEW(p,c1,...,cn) (D.20).
- For DISPOSE(p,k1...,km), it is an error if the variable had been created using the form NEW(p,c1,...,cn) and m is not equal to n (D.21).
- For DISPOSE(p,k1,...,km), it is an error if the variants in the variable identified by the pointer value of p are different from those specified by the case constants k1,...,km (D.22).
- It is an error if a variable created using the second form of NEW is accessed by the identified variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter (D.25).
- For PACK, it is an error if any of the components of the unpacked array are both undefined and accessed (D.27).
- For UNPACK, it is an error if any of the components of the packed array are undefined (D.30).
- An expression denotes a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use is an error (D.43).

- It is an error if the result of an activation of a function is undefined upon completion of the algorithm of the activation (D.48).

- On writing to a textfile, the values of TotalWidth and FracDigits are greater than or equal to 1; it is an error if either value is less than 1 (D.58).

H. I/O PROGRAMMING EXAMPLES

The following programs provide examples to help familiarize users with the nature of Pascal I/O.

Example 1:

```
PROGRAM intcpy(infil, OUTPUT);
   (* THIS PROGRAM COPIES A TEXT FILE OF INTEGERS ( 1 PER LINE ) FROM
      COS LOCAL DATASET INFIL TO THE STANDARD OUTPUT FILE *)

   VAR
      i : INTEGER;
      buffer, wdcnt, status : INTEGER;
      dsname : ALFA;
      infil : TEXT;

   BEGIN

   RESET( infil);

   WHILE NOT EOF( infil) DO
      BEGIN

      READLN( infil, i);

      WRITELN( OUTPUT, i);

      END;

   END.
```

Example 2:

```
PROGRAM strcpy( infil, outfil);
   (* THIS PROGRAM COPYS LINES OF TEXT FROM COS LOCAL DATASET
      INFIL TO COS LOCAL DATASET BLEEM. IF LOCAL DATASET BLEEM
      ALREADY CONTAINS SOMETHING, IT IS OVERWRITTEN *)

   VAR
      str : ARRAY[1..132] OF CHAR;
      i, length : 0..132;
      infil, outfil : TEXT;
```

```
BEGIN

CONNECT( outfil, 'BLEEM    ');
REWRITE( outfil);

RESET( infil);
(* NOTE THAT INFIL WAS AUTOMATICALLY CONNECTED WITH COS LOCAL
   DATASET INFIL, SINCE IT APPEARED IN THE PROGRAM HEADER.

   OUTFIL WOULD HAVE BEEN SIMILARLY CONNECTED HAD WE NOT
   OVERRIDDEN THE CONNECTION WITH THE CONNECT PREDEFINED
   PROCEDURE *)

WHILE NOT EOF( infil) DO
   BEGIN

   length := 0;

   WHILE NOT EOLN( infil) DO
      BEGIN

      READ(infil, str[ length + 1]);
      length := length + 1;

      END;

   READLN( infil);

   FOR i := 1 TO length DO
      WRITE( outfil, str [i]);

   WRITELN( outfil);

   END;

END.
```

INDEX

INDEX

READER COMMENT FORM

Pascal Reference Manual SR-0060 B

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

DATE _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

Attention:
PUBLICATIONS

STAPLE

READER COMMENT FORM

Pascal Reference Manual SR-0060 B

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

DATE _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

Attention:
PUBLICATIONS

STAPLE