DOMAIN Pascal Language Reference

Order No. 000792
Revision 04

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: September, 1981
Latest Printing: January, 1987
Updated: December, 1986

This document was produced using the Interleaf Workstation Publishing Software (WPS). Interleaf and WPS are trademarks of Interleaf, Inc.

Preface

The *DOMAIN Pascal Language Reference* explains how to code, compile, bind, and execute DOMAIN Pascal programs.

We've organized this manual as follows:

Chapter 1	Introduces DOMAIN Pascal and provides an overview of its extensions.
Chapter 2	Defines DOMAIN Pascal building blocks (like the length of an identifier) and describes the structure of the main program.
Chapter 3	Explains all the DOMAIN Pascal data types.
Chapter 4	Contains alphabetized listings describing all the functions, procedures, statements, and operators that you can use in the code portion of a program.
Chapter 5	Explains how to declare and call procedures and functions.
Chapter 6	Details compiling, binding, debugging, and executing.
Chapter 7	Describes how you can break your program into two or more separately–compiled modules (which can be in DOMAIN Pascal, DOMAIN FORTRAN, or DOMAIN C).
Chapter 8	Contains an overview of the I/O resources available to DOMAIN Pascal programmers.
Chapter 9	Covers errors and warnings and how to handle them.
Appendix A	Is a table of DOMAIN Pascal reserved words and predeclared identifiers.
Appendix B	Is an ASCII table.
Appendix C	Describes DOMAIN Pascal's extensions to ISO/ANSI standard Pascal.
Appendix D	Describes DOMAIN Pascal's deviations from ISO/ANSI standard Pascal.
Appendix E	Describes built–in routines available for systems programmers.

Audience

We wrote this manual to serve programmers at a variety of levels of Pascal expertise. Our goal is to keep the writing as simple as possible, but to assume that you know the fundamentals of Pascal programming. If you are totally inexperienced in a block–structured language like Pascal or PL/I, you probably should study a Pascal tutorial before using this manual. If you have a little experience with a block–structured language, you will probably benefit most by experimenting with the many examples we provide (particularly in Chapter 4). If you are an expert Pascal programmer, turn to Appendix C first for a list of our extensions to standard Pascal.

Summary of Technical Changes

This manual describes the DOMAIN Pascal compiler available with Software Release 9.5 and supersedes all earlier documents. The last update to the DOMAIN Pascal manual was at SR9.0. The following list describes some of the features added to DOMAIN Pascal since SR9.0:

- The compiler reports many new error and warning messages.
- The new built-in functions **max** and **min** return the larger and smaller, respectively, of two values.
- The new built-in procedure **discard** explicitly discards the computed value of an expression.
- DOMAIN Pascal now supports the ISO/ANSI standard Pascal procedures **pack** and **unpack.**
- New compiler optimizations eliminate variable assignments that are never used.
- A new compiler directive, **%slibrary,** allows the use of precompiled library include files. Also, the new compiler option **-slib** allows you to precompile a file so that **%slibrary** can use it.
- The **-opt** compiler option now allows you to select a predetermined level of optimization for your program. You can select from level 0 (which is equivalent to **-nopt**) to level 3.
- The **-cpu** compiler option now accepts the following additional arguments for targeting the run-time CPU: 560 | 330 | 90 | 570 | 580 | 3000.
- Three new routine options are available: **nosave, noreturn,** and **d0_return.**
- The **volatile** attribute has been changed so that the attribute no longer is inherited by an entire **record** for which one field is declared **volatile.**
- Enumerated type variables can now have a maximum of 2048 elements. The old limit was 256.
- The compiler no longer requires that DOMAIN Pascal source filenames end with **.pas.**
- A new optional parameter for the **open** procedure allows you to specify a buffer size.
- A new systems programming function, **set_sr,** saves the current value of the hardware status register and then inserts a new value.

Revisions to Manual

A number of changes have been made to this manual for this revision. To help you find the revisions, change bars like the one next to this paragraph appear on the revised pages indicating where new information has been added or existing information has been revised.

Related Manuals

- *Getting Started With Your DOMAIN System* (002348) explains the fundamentals of the DOMAIN system.
- The *DOMAIN System User's Guide* (005488) provides more detailed information on the DOMAIN system.
- The *DOMAIN C Language Reference* (002093) describes the DOMAIN implementation of the C language.

- The *DOMAIN FORTRAN Language Reference* (000530) describes the DOMAIN implementation of FORTRAN.
- The *DOMAIN System Command Reference* (002547) describes the Shell and Display Manager (DM) concepts and commands.
- The *DOMAIN System Call Reference* (007196) describes the system service routines provided by the operating system, and explains how to call these routines from user programs.
- *Programming With General System Calls* (005506) covers writing DOMAIN Pascal programs that use stream calls and many other important system calls.
- The *DOMAIN Binder and Librarian Reference* (004977) describes how to use the bind utility to link object modules and the librarian utility to create library files.
- The *DOMAIN Language Level Debugger Reference* (001525) describes the high–level language debugger, DEBUG.
- The *DOMAIN Software Engineering Environment™ (DSEE) Reference* (003016) describes the DOMAIN DSEE product.
- The *DOMAIN®/Dialogue™ User's Guide* (004299) describes the DOMAIN/Dialogue product.

Pascal Tutorials

- Jensen, K. and N. Wirth, revised by Mickel, A. and J. Miner. *Pascal User Manual and Report.* Third Edition. Springer–Verlag, New York: 1985.
- Grogono, Peter. *Programming in Pascal.* Revised Edition. Reading, Massachusetts: Addison–Wesley, 1980.
- Cooper, D., and M. Clancy. *Oh! Pascal!* New York: WW Norton, 1982.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software–related comments, and the Reader's Response form for documentation comments. By using these formal channels you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference*. Refer to the **crucr** (CREATE_USER_CHANGE_REQUEST) Shell command description. You can view the same description on–line by typing:

```
$ help crucr
```

For your documentation comments, we've included a Reader's Response form at the back of each manual.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

boldface	Boldface words or characters in descriptions of statement or keyword syntax represent the keywords that you must use literally.
lowercase	Lowercase words or characters in descriptions of statement or keyword syntax represent values that you must supply.
italics	Italicized words in descriptions of statement or keyword syntax represent optional items.
`typewriter`	Typewriter font is used for examples and for variable, program, and routine names within text.
Extension	The word "Extension" used in a section heading indicates that all the information described in the section is a DOMAIN Pascal extension to the ISO standard.
color	Words printed in color denote user input to a program example or on a command line. Color also indicates an extension to ISO standard Pascal.
{ Braces }	Braces enclose comments in a program or command.
|	A vertical bar separates items in a list of choices.
<>	Angle brackets enclose the name of a key on the keyboard; e.g.,<RETURN>.
CTRL/Z	The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down <CTRL> while typing the character.
. . .	Horizontal ellipsis points in descriptions of statements and keyword syntax indicate that the preceding item may be repeated one or more times. In program examples, horizontal ellipsis points mean irrelevant parts of the example have been omitted.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

Contents

Illustrations

Tables

Chapter 1

Introduction

This manual describes DOMAIN Pascal, which is our version of ISO standard Pascal (ISO 7185–1982 Level 0). Niklaus Wirth developed Pascal in the late 1960s to serve primarily for teaching and for realizing efficient, reliable, and portable programs. Since then, Pascal has become a widely used commercial language. Pascal contains a small set of constructs, and it is fairly easy to learn.

You should be somewhat familiar with Pascal before attempting to use this manual. If you are not, please consult a good Pascal tutorial. (We've listed some good tutorials in the Preface.) If you are somewhat familiar with Pascal or if you are expert in a highly block-structured language such as PL/I, then you should be able to write programs in DOMAIN Pascal after reading this manual.

1.1 A Sample Program

The best way to get started with DOMAIN Pascal is to write, compile, and execute a simple program. Figure 1-1 shows a simple program that you can use to get started. You are welcome to type in this program yourself, but you can use the **getpas** utility (described in the next section) to save some typing.

```
PROGRAM getting_started;
{A simple program to try out.}

VAR
 x : integer16;
 y : integer32;

BEGIN
 write('Enter an integer -- ');
 readln(x);
 y := x * 2;
 writeln;
 writeln(y:1, ' is twice ', x:1);
END.
```

Figure 1-1. Sample Program

Suppose you store the program in file easy.pas. (Although it is not required that the filename end with the **.pas** extension, we recommend its use so that you can readily identify Pascal source programs.)

To compile a program, simply enter the Shell command **pas** followed by the filename. If you do use the **.pas** extension, you can include or omit that extension at this step. DOMAIN Pascal doesn't care which way you type it. For example, to compile easy.pas, you can enter either of the following commands:

```
$ pas easy
```

or

```
$ pas easy.pas
```

The compiler creates an executable object in filename easy.bin. To execute this object you merely enter its name; for example:

```
$ easy.bin
Enter an integer -- 15

30 is twice 15
```

1.2 On-Line Sample Programs

Many of the programs from this manual are stored on-line, along with sample programs from other DOMAIN manuals. These programs illustrate features of the Pascal language, and demonstrate programming with DOMAIN graphics calls and system calls.

The sample programs are stored in one master file (to conserve disk space). To extract a sample program from this master file, you must execute the **getpas** program. **Getpas** prompts you for the name of the sample program and the pathname of the file to which you want to copy it.

Before you use **getpas** for the first time, you must create the proper links. If the on-line examples are stored on your node, you only need to create the following single link:

```
$ crl ~com/getpas /domain_examples/pascal_examples/getpas
```

However, if the on-line examples are stored on another node, you need to create the following two links:

```
$ crl /domain_examples/pascal_examples @
$_  //othernode/domain_examples/pascal_examples
$ crl ~com/getpas //othernode/domain_examples/pascal_examples/getpas
```

where othernode is the name of the disk on which the examples are stored. (The @ symbol on the first line of this example is the AEGIS shell's continuation character. You must include it to continue a command on another line. In a DOMAIN/IX shell, the backslash (\) is the continuation character.)

Once you create the appropriate links, you can execute the **getpas** program. Here is a **getpas** demonstration:

```
$ getpas
Welcome to getpas -- please wait, initializing master file


Enter the name of the program you want to retrieve, or
enter HELP to get a list of available programs, or
enter QUIT to exit -- getting_started

Program getting_started has been found.

What pathname would you like to store getting_started in?
  (default is getting_started.pas) -- easy.pas
Program getting_started written to output file easy.pas


Enter the name of the program you want to retrieve, or
enter HELP to get a list of available programs, or
enter QUIT to exit -- quit
```

Getpas warns you if you try to write over an existing file.

As stated, many of this manual's sample programs are available from **getpas.** When one is, the name of the program appears just after the program listing. You simply have to run **getpas** and request the appropriate `program_name`.

1.3 Overview of DOMAIN Pascal Extensions

DOMAIN Pascal supports many extensions to ISO/ANSI standard Pascal. The purpose of this section is to provide an overview of these extensions. For a complete list of all the extensions, see Appendix C. From Chapter 2 to Chapter 5, all extensions to the standard are marked in color like this or are noted explicitly in text as an extension. For a list of all omissions from the standard, see Appendix D.

Naturally, the more you take advantage of DOMAIN Pascal extensions, the less portable your code will be. Therefore, if you are very concerned with portability, you should avoid using the following features.

1.3.1 Extensions to Program Organization

Chapter 2 describes the organization of a DOMAIN Pascal program contained within one file. Following is an overview of the extensions described within the chapter. You can:

- Specify an underscore (_) or dollar sign ($) in an identifier.
- Specify integers in any base from 2 to 16.
- Specify comments in three ways.
- Specify that the compiler assign the code or data in your program to nondefault named sections.
- Declare the **label**, **const**, **type**, and **var** declaration parts in any order.
- Declare a **define** part, which is in addition to **label**, **const**, **type**, and **var** declaration parts. (Chapter 7 details this extension more fully.)
- Use constant expressions when declaring constants, as long as the components of the expressions are constants.
- Use both identifiers and integers as labels.

1.3.2 Extensions to Data Types

Chapter 3 describes the data types supported by DOMAIN Pascal. DOMAIN Pascal supports the following extensions that allow you to:

- Specify two additional pointer types. The first is a special pointer to procedures and functions. The second is a universal pointer type that will hold a pointer to a variable of any data type.
- Initialize static variables in the **var** declaration part of your program.
- Group variables into named sections for better runtime performance.
- Specify variable and type attributes that let you better control compiler optimization.

1.3.3 Extensions to Code

Chapter 4 describes the action portion of your program. DOMAIN Pascal supports the following extensions to executable statements:

- An **addr** function that returns the address of a specified variable
- Bit operators or functions for bitwise *and*, *not*, *or*, and *exclusive or* operations
- Three bit-shift functions (**rshft**, **arshft**, and **lshft**)

- An **open** procedure for opening files and a **close** procedure for closing files
- Many compiler directives that enable features like include files and conditional compilation
- A **discard** procedure for explicitly discarding an expression's value and so suppressing some compiler optimizations
- An **exit** statement for jumping out of the current loop
- A **find** procedure for locating a specified element in a file
- A **firstof** and a **lastof** function for returning the first and last possible value of a specified data type
- Some additional capabilities for the **if** statement
- An **in_range** function for determining whether a specified value is within the defined range of an integer subrange or enumerated type
- A **max** and a **min** function for finding the greater and lesser of two specified expressions
- A **next** statement for skipping over the current iteration of a loop
- A **replace** procedure that allows you to modify an existing element in a file
- A **return** statement for causing a premature return to a calling procedure or function
- A **sizeof** function for returning the size (in bytes) that a specified data type requires in storage
- Additional type transfer functions that transform the data type of a variable or expression into some other data type
- Some additional capabilities for the **with** statement

1.3.4 Extensions to Routines

Chapter 5 describes procedures and functions (routines). This chapter documents extensions that allow you to:

- Specify the direction of parameter passing with the special **in**, **out**, and **in out** keywords.
- Use the **univ** keyword to suppress parameter type checking.
- Specify routine attribute clauses and routine options clauses to control how the compiler processes a routine.

1.3.5 Extensions to Program Development

Chapter 6 explains how to compile, bind, debug, and execute your program. Program development tools are an implementation–dependent feature of a Pascal implementation; that is, there is no standard for these tools.

1.3.6 External Routines and Cross–Language Communication

Chapter 7 explains how to write a program that accesses code or data written in another separately–compiled module or library. It also describes how to access routines written in DOMAIN FORTRAN or DOMAIN C. The entire chapter describes implementation–dependent features.

1.3.7 Extensions to I/O

Chapter 8 describes input and output from a DOMAIN Pascal programmer's point of view. DOMAIN Pascal supports all the standard I/O procedures. In addition, it supports the **open**, **close**, **find**, and **replace** procedures. As a further extension to the standard, DOMAIN Pascal permits you to easily access the operating system's I/O and formatting system calls.

1.3.8 Errors

Chapter 9 lists compiletime and runtime error messages and explains how to deal with them. Error messages are an implementation–dependent feature of Pascal.

Chapter 2

Blueprint of a Program

This chapter describes the building blocks and organization of a DOMAIN Pascal program.

2.1 Building Blocks of DOMAIN Pascal

When describing a language implementation, it is customary to describe the basic building blocks or elements of that implementation. This section defines identifiers, integers, real numbers, comments, and strings. It also explores case–sensitivity and spreading source code across multiple lines.

2.1.1 Identifiers

In this manual, the term "identifier" refers to any sequence of characters that meets the following criteria:

- The first character is a letter
- The remaining characters are any of the following:

  A..Z and a..z
  0..9
  _ (underscore)
  $ (dollar sign)

An identifier can be any length, but DOMAIN Pascal ignores any characters beyond the 32nd. Therefore, DOMAIN Pascal cannot distinguish between the following two identifiers (the column ruler at the top is there to help you with character counting):

```
{ ruler:          1         2         3     }
{        1234567890123456789012345678901234   }


         Accounts_receivable_kansas_city_kn
         Accounts_receivable_kansas_city_mo
```

Identifiers are case-insensitive.

2.1.2 Integers

The first character of an integer must be a positive sign (+), a negative sign (–), or a digit. Each succeeding character must be a digit. (See the "Integers" section in Chapter 3 for a description of the range of various integer data types.)

An unsigned integer must begin with a digit. Each succeeding character must be a digit.

Note that Pascal prohibits two consecutive mathematical operators. If you want to divide 9 by negative 3, you might be tempted to use the following expression:

```
9 DIV -3
```

However, this produces an error, since Pascal interprets the negative sign as the subtraction operator (and that makes two mathematical operators in a row). Where the sign of an integer can be confused for an addition or subtraction operator, enclose the integer with parentheses. Thus, the correct expression for 9 divided by negative 3 is:

```
9 DIV (-3)
```

Pascal assumes a default of base 10 for integers. If you want to express an integer in another base, use the following syntax:

base#value

For base, enter an integer from 2 to 16. For value, enter any integer within that base. If the base is greater than 10, use the letters A through F (or a through f) to represent digits with the values 10 through 15.

For example, consider the following integer constant declarations:

```
half_life  := 5260;      /* default     (base 10) */
hexograms  := 16#1c6;    /* hexadecimal (base 16) */
luck       := 2#10010;   /* binary      (base 2)  */
wheat      := 8#723;     /* octal       (base 8)  */
```

2.1.3 Real Numbers

DOMAIN Pascal supports the standard Pascal definition of a real number, which is

integer.*unsigned_integer*E*integer*

In other words, a valid real number may or may not contain a decimal point. If the number contains a decimal point, you must specify at least one digit to the left of the decimal point and at least one digit to the right of the decimal point. To express expanded notation (powers of 10), use the letter e or E followed by the exponent; for example:

```
5.2      means  +5.2
5.2E0    means  +5.2
-5.2E3   means  -5200.0
5.2E-2   means  +0.052
```

Compare the right and wrong way for writing decimals in your program:

```
.5    {wrong}
0.5   {right}
5E1   {right}
```

Note that although using *.5* in your source code causes an error at compiletime, entering *.5* as input data to a real variable does not cause an error at runtime.

2.1.4 Comments

You can specify comments in any of the following three ways:

```
{ comment }
(* comment *)
"comment"
```

(The spaces before and after the comment delimiters are for clarity only; you don't have to include these spaces. If you use a compiler directive within comment delimiters you *cannot* use spaces; see the listing for "Compiler Directives" in Chapter 4 for details.) For example, here are three comments:

```
{ This is a comment. }
(* This is a comment. *)
"This is a comment."
```

Unlike standard Pascal, the comment delimiters of DOMAIN Pascal must match. For example, a comment that starts with a left brace doesn't end until the compiler encounters a right brace. Therefore, you can nest comments, for example:

```
{ You can (*nest*) comments inside other comments. }
```

Standard Pascal does not permit nested comments. If you want to use unmatched comment delimiters, as standard Pascal allows, you must compile with the **-iso** switch. Chapter 6 describes this switch.

The DOMAIN Pascal compiler ignores the text of the comment, and interprets the first matching delimiter as the end of the comment.

Use quotation marks to set off comments only if you are converting existing applications to the DOMAIN system. In all other programs, you should use either of the other two methods.

Note that Pascal comments can stretch across multiple lines; for example, the following is a valid comment:

```
{ This is a comment
  that stretches across
  multiple lines. }
```

NOTE: You can use the **-comchk** compiler option (described in Chapter 6) to warn you if a new comment starts before an old one finishes. This option can help you find places where you forgot to close a comment.

2.1.5 Strings

We refer to strings throughout this manual. In DOMAIN Pascal, a string is a sequence of characters that starts and ends with an apostrophe. Unlike an identifier, you can use any printable character within a string. Here are some sample strings:

```
'This is a string.'
'18'
'b[2~{q^%pl'
'can''t'
```

To include an apostrophe in a string, write the apostrophe twice; for example:

```
'I can''t do it.'
'Then don''t try!'
```

NOTE: Within a string, DOMAIN Pascal treats the comment delimiters as ordinary characters rather than as comment delimiters.

2.1.6 Case Sensitivity

DOMAIN Pascal, like standard Pascal, is case–insensitive to keywords and identifiers (i.e., variables, constants, types, and labels), but case–sensitive to strings. That is, DOMAIN Pascal makes no distinction between uppercase and lowercase letters except within a string. For example, the following three uses of the keyword **begin** are equivalent:

```
BEGIN
begin
Begin
```

However, the following two character strings are not equivalent:

```
'The rain in Spain';
'THE RAIN IN SPAIN';
```

2.1.7 Spreading Source Code Across Multiple Lines

In Pascal, you can start a statement or declaration at any column and spread it over as many lines as you want. However, note that you cannot split a token (keyword, identifier, or string) across a line. For example, consider the **writeln** statement which can take character strings as an argument. The following use of **writeln** is wrong because it splits the string across a line:

```
WRITELN('This is an uninteresting
        long string');
```

Instead, the line should appear this way:

```
WRITELN('This is an uninteresting long string');
```

NOTE: By default, any text file you **open** for reading can have a maximum of 256 characters per line. You can specify an optional buffer size when you **open** the file, however, to change that default.

2.2 Organization

You can write a DOMAIN Pascal program in one file or across several files. This section explains the proper structure for a program that fits into one file. Chapter 7 details the structure for a program that is spread over several files.

A DOMAIN Pascal program takes the format shown in Figure 2–1.

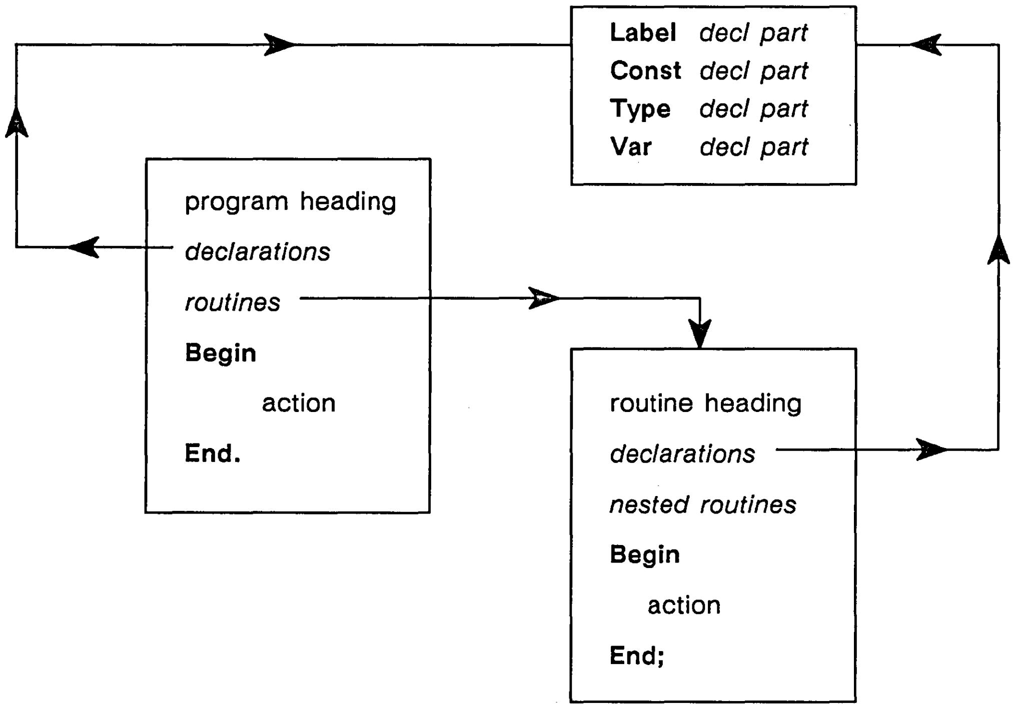

Figure 2–1. Format of Main Program in DOMAIN Pascal

Note that routines are themselves declarations.

Figure 2–2 is a labeled program designed to help you understand the structure of a DOMAIN Pascal program.

```
PROGRAM labeled;                                    {program heading}

{Start of the declaration part of the main program.      }
{These declarations will be global to the entire program.}
   LABEL                                     {LABEL declaration part}
      finish;

   CONST                                     {CONST declaration part}
      axiom = 'Clarity is wonderful!';

   TYPE                                      {TYPE declaration part}
      flavors = (mint, lime, orange, beige);

   VAR                                       {VAR declaration part}
      x, y, z : integer;
      ice_cream : flavors;
{End of the declaration part of the main program.}



{Start of the roots procedure.}
   Procedure roots;                          {routine heading}

{Start of the declaration part of roots.}
{These declarations will be local to roots.}
   VAR                                       {VAR declaration part}
      q : real;
{End of the declaration part of roots.}

   BEGIN           {Start of the action part of roots.}
      write('Enter a number -- '); readln(q);
      writeln('The square root of ',q:-1, ' is ',sqrt(q));
   END;            {End of the action part of roots.}
{End of the roots procedure.}



BEGIN              {Start of the action part of the main program.}
   writeln(axiom);
   x := 5;  y := 7;  z := x + 5;
   if z > 100 then goto finish;
   for ice_cream := mint to beige do
      writeln(ice_cream);
   roots;
finish:
END.               {End of the action part of the main program.}
```

Figure 2-2. Labeled Main Program

The following subsections detail the parts of a program.

2.2.1 Program Heading

Your program must contain a program heading. The program heading has the following format:

program name *(file_list), code_section_name, data_section_name;*

In DOMAIN Pascal, as in standard Pascal, you must supply a name for the program. Name must be an identifier. This identifier has no meaning within the program, but is used by the binder, the librarian, and the loader. (See the *DOMAIN Binder and Librarian Utility* manual for details on these utilities.)

In standard Pascal you can supply an optional *file_list* to the program heading. The *file_list* specifies the external files (including standard **input** and **output**) that you are going to access from the program. However, unlike standard Pascal, the *file_list* in a DOMAIN Pascal program has no effect on program execution; the compiler ignores it. (For details on I/O, see Chapter 8.)

Code_section_name and *data_section_name* are optional elements of the program heading. Use them to specify the names of the sections in which you want the compiler to store your code and data. A section is a named contiguous area of memory in which all entities share the same attributes. (See the *DOMAIN Binder and Librarian Reference* for details on sections and attributes.) By default, DOMAIN Pascal assigns all the code in your program to the PROCEDURE$ section and all the data in your program to the DATA$ section. To assign your code and data to nondefault sections, specify a *code_section_name* and a *data_section_name*.

NOTE: In addition to nondefault code and data section names for the entire program, you can also specify a nondefault section name for a procedure, a function, or a group of variables. See Chapter 5 for an explanation of how to assign section names to procedures and functions, and see Chapter 3 to learn how to assign section names to groups of variables.

Let's now consider some sample program headings. Despite the options available, most DOMAIN Pascal program headings can look as simple as the following:

```
Program trapezoids;
```

Those of you desiring to write standard Pascal programs will also probably want to supply a *file_list* as in the next example:

```
Program trapezoids (input, output, datafile);
```

Finally, those of you wanting to capitalize on certain runtime features may wish to define your own section names. For example, if you want the compiler to store the code into section mycode and the data into section mydata, you would issue the following program heading:

```
Program trapezoids, mycode, mydata;
```

2.2.2 Declarations

The declarations part of a program is optional. It can consist of zero or more **label** declaration parts, **const** declaration parts, **type** declaration parts, and **var** declaration parts. DOMAIN Pascal allows you to specify these parts in any order.

2.2.2.1 Label Declaration Part

You define labels in the **label** declaration part. A label has only one purpose -- to act as a target for a **goto** statement. In other words, the statement

```
GOTO 40;
```

only works if you have defined 40 as a label.

The format for a **label** declaration part is

```
label
        label1, ... labelN;
```

A label is either an identifier or an unsigned integer. If there are multiple labels, you must separate them with commas. Remember, though, to put a semicolon after the final label.

For example, the following is a sample label declaration:

```
LABEL
   40, reprompt, finish, 9999;
```

See Chapter 4 for a description of the **goto** statement.

2.2.2.2 Const Declaration Part

You define constants in the **const** declaration part. A constant is a synonym for a value that will not (and cannot) change during the execution of the program. The **const** declaration part takes the following syntax:

```
const
        identifier1 = value1;
              .         .
              .         .
              .         .
        identifierN = valueN;
```

An identifier is any valid DOMAIN Pascal identifier. A value must be a real, integer, string, char, or set constant expression. Value can also be the pointer expression **nil**.

For example, here is a sample **const** declaration part:

```
CONST
   pi = 3.14;                                 {A real number.}
   cup = 8;                                   {An integer.}
   key = 'Y';                                 {A character.}
   blank = ' ';                               {A character.}
   words = 'To be or not to be';              {A string.}
   vowels = ['a', 'e', 'i', 'o', 'u'];        {A set.}
   ptr1 = nil;                                {A pointer.}
```

The preceding sample involves simple expressions; however, you can also specify a more complex expression for value. Such an expression can contain the following types of terms:

- A real number, an integer, a character, a string, a set, a Boolean, or **nil**
- A constant that has already been defined in the **const** declaration part (note that you cannot use a variable here)
- Any predefined DOMAIN Pascal function (e.g., **chr**, **sqr**, **lshft**, **sizeof**, but only if the argument to the function is a constant)
- A type transfer function

You can optionally separate these terms with any of the following operators:

Operator	Data Type of Operand
+, –, *	Integer, real, or set
/	Real
mod, div, !, &, ~	Integer

Chapter 4 describes these operators.

For example, the following **const** declaration part defines eight constants:

```
CONST
   x = 10;
   y = 100;
   z = x + y;
   current_year = 1994;
   leap_offset  = (current_year mod 4);
   bell         = chr(7);
   pathname     = '//et/go_home';
   pathname_len = sizeof(pathname);
```

2.2.2.3 Type Declaration Part

Chapter 3 details the many predeclared data types DOMAIN Pascal supports. In addition to these Pascal-defined data types, you can create your own data types in the **type** declaration part. After creating your own data type, you can then declare variables (in the **var** declaration part) that have these data types. The format for a **type** part is as follows:

```
type
      identifier1 = data_type1;
          .          .
          .          .
          .          .
      identifierN = data_typeN;
```

An identifier is any valid DOMAIN Pascal identifier. A data_type is any predeclared DOMAIN Pascal data type (like **integer** or **real**), any data type that you create, or the identifier of a data type that you created earlier in the **type** declaration part. For example, here is a sample **type** declaration part:

```
TYPE
   long = integer32;         {A predeclared DOMAIN Pascal data type.}
   student_name = array[1..20] of long;   {A user-defined data type.}
   colors = (magenta, beige, mauve);      {A user-defined data type.}
   hue = set of colors;                   {A user-defined data type.}
   table = array[magenta..mauve] of real; {A user-defined data type.}
```

2.2.2.4 Var Declaration Part

Declare variables in the **var** declaration part. A variable has two components -- a name and a data type. The format for the **var** declaration part is:

```
var
     identifier_list1 : data_type1;
          .              .
          .              .
          .              .
     identifier_listN : data_typeN;
```

An identifier_list consists of one or more identifiers separated by commas. Each identifier in the identifier_list has the data type of data_type. Data_type must be one of these:

- A predeclared DOMAIN Pascal data type
- A data type you declared in the **type** declaration part
- An anonymous data type (that is, a data type you declare for the variables in this identifier_list only)

For example, consider the following **type** declaration part and **var** declaration part:

```
TYPE
   names = array[1..20] of char;
   colors = (red, yellow, blue);

   VAR
      counter, x, y : integer;      {integer is a predeclared DOMAIN Pascal
                                     data type.}
      angles        : real;         {real is a predeclared DOMAIN Pascal
                                     data type.}
      a_letter      : char;         {char is a predeclared DOMAIN Pascal
                                     data type.}
      couch_colors  : colors;       {colors is defined in the TYPE part.   }
      evil          : boolean;      {boolean is a predeclared DOMAIN Pascal
                                     data type.
      mystery_guest : names;        {names is defined in the TYPE part.    }
      seniors       : 67..140;      {An anonymous subrange data type.      }
      pet           : (cat, dog);   {An anonymous enumerated data type.    }
```

In the preceding example, note that `counter`, `x`, and `y` are three variables that have the same data type (**integer**).

2.2.2.5 Define Declaration Part -- Extension

In addition to the **const**, **type**, **var**, and **label** declaration parts of standard Pascal, DOMAIN Pascal also supports an optional **define** declaration part, which is described in Chapter 7.

2.2.3 Routines

A program can contain zero or more routines. There are two types of routines in DOMAIN Pascal: procedures and functions. A routine consists of three parts: a routine heading, an optional declaration part, and an action part.

2.2.3.1 Routine Heading

Routine headings take the following format:

attribute_list **procedure** name (*parameter_list); routine_options*;

or

attribute_list **function** name (*parameter_list*) : data_type; *routine_options*

where:

- o *attribute_list* is optional. Inside the *attribute_list*, you can specify nondefault section names for the routine's code and data. For a description, see Chapter 5.
- name is an identifier. You call the routine by this name.
- *parameter_list* is optional. It is here that you declare the names and data types of all the parameters that the routine expects from the caller. See Chapter 5 for details on the *parameter_list.*
- data_type is the data type of the value that the function returns. The difference between a procedure and a function is that the name of a procedure is simply a name, but the name of a function is itself a variable with its own data_type. You must assign a value to this variable at some point within the action part of the function. (It is an error if you don't.) You cannot assign a value to the name of a procedure. (It is an error if you do.)
- o *routine_options* is an optional element of the routine heading. You can specify characteristics of the routine such as whether or not it can be called from another routine. Chapter 5 describes the *routine_options*.

2.2.3.2 Declaration Part of a Routine

The optional declaration part of a routine follows the same rules (with one exception) as the optional declaration part under the program heading. The constants, data types, variables, and labels are local to the routine declaring them and to any routines nested within them. (See the "Global and Local Variables" and "Nested Routines" sections at the end of this chapter for details.)

The one difference between the declaration part of a routine and the declaration part of the main program is that the declaration part of a routine cannot contain a **define** declaration part.

2.2.3.3 Nested Routines

You can optionally nest one or more routines within a routine. See the "Nested Routines" section at the end of this chapter for details.

2.2.3.4 Action Part of a Routine

The action part of a routine starts with the keyword **begin** and finishes with the keyword **end**. Between **begin** and **end** you supply one or more DOMAIN Pascal statements. (See Chapter 4 for a description of DOMAIN Pascal statements.) You must place a semicolon after the final **end** in a routine. For example, consider the following sample action part of a routine:

```
BEGIN
   x := x * 100;
   writeln(x);
END;
```

2.2.4 Action Part of the Main Program

The action part of the main program is almost identical to the action part of a routine. Both start with **begin**, both finish with **end**, and both contain DOMAIN Pascal statements in between. The only difference is that you must place a period (rather than a semicolon) after the final **end** in the main program. For example, consider the following sample action part of the main program:

```
BEGIN
   x := x * 100;
   writeln(x);
END.
```

2.3 Global and Local Variables

The declarations in the declaration part of the main program are global to the entire program. The declarations in the declaration part of a routine are local to that routine (assuming no nesting). For example, consider the following program. In it, variable g is global and variable l is local to procedure add100;

```
Program scope;
VAR
   g : integer16;

   Procedure add100;
   VAR
      l : integer16;
   BEGIN
      l := g + 100;   {Variable l is accessible within this procedure only,}
                      {while g is global and so is accessible anywhere.    }
   END;



BEGIN
   g := 10;           {Variable g is accessible because it is global. }
   add100;            {Call the procedure.                            }

                      {Variable l is not accessible here because it is}
                      {local to procedure add100.                     }
END.
```

What happens when you specify a local variable with the same name as a global variable? To answer this question, here are two more programs. In the program on the left (`global_example`), `x` is declared as a global variable. In the program on the right (`local_example`), `x` is declared twice. The first declaration specifies `x` as a global variable. The second declaration declares `x` as local to procedure `convert`.

```
Program global_example;

VAR {global declarations}
   x : integer16;


   PROCEDURE convert;


   BEGIN
      x := -10;
      writeln('In convert, x=',x:1);
   END;


BEGIN  {main}
   x := +10;
   convert;
   writeln('In main, x=',x:1);
END.
```

```
Program local_example;

VAR {global declarations}
   x : integer16;


   PROCEDURE convert;
   VAR {local declarations}
      x : integer16;
   BEGIN
      x := -10;
      writeln('In convert, x=',x:1);
   END;


BEGIN {main}
   x := +10;
   convert;
   writeln(In main, x=', x:1);
END.
```

If you execute these programs, you get the following results:

Execution of global_example

```
In convert, x= -10
In main, x= -10
```

Execution of local_example

```
In convert, x= -10
In main, x=  10
```

In program `local_example`, within procedure `convert`, the declaration of the local variable `x` overrides the global declaration of `x`. Within `convert`, the fact that the local variable and the global variable have the same name (`x`) prevents procedure `convert` from accessing the global variable `x` at all.

Both programs are available on-line and are named `global_example` and `local_example`.

2.4 Nested Routines

A nested routine is a routine that is declared inside another routine. A nested routine can access any declared object (label, constant, type, or variable) in a routine outside it. The reverse is not true; that is, a routine cannot access an object in a routine nested inside it. Thus, the purpose of nesting routines is to create a hierarchy of access. You might view declared objects in the following way:

- Global to the entire program.
- Local to a single routine.
- Local to the routine it is defined in and to all routines nested within it (i.e., neither truly local nor truly global). This is termed an "intermediate level" object.

Note that the main program is itself a routine, and that all routines are nested at least one level inside it. A routine can call any routine nested one level inside it, but cannot explicitly call any routine nested two or more levels inside it. A routine can also call any routine at its level or outside it, though a routine cannot explicitly call the main program.

For example, consider the program in Figure 2-3. Procedure `one` is nested inside the main program. Procedures `twoa` and `twob` are both nested inside procedure `one`. The most-nested procedures (`twoa` and twob) can access the most variables. The least-nested procedure (the main program) can access the least number of variables.

```
Program nesting_example;

VAR
   g : integer16;

   procedure one;
   VAR
      l : integer16;

      procedure twoa;
      VAR
         n1 : integer16;
      BEGIN {twoa}
      {can access g, l, and n1.}
         n1 := l + g + 500;
      END;  {twoa}


      procedure twob;
      VAR
         n2 : integer16;
      BEGIN  {twob}
      {can access g, l, and n2.}
         n2 := l + g + 1000;
      END;   {twob}


   BEGIN {one}
   {can access g and l.}
      l := g + 10;
      twob;
   END;  {one}

BEGIN  {main program}
{can only access g.}
   g := 1;
   g := g * 2;
   one;
END.   {main program}
```

Figure 2-3. Nesting Example

Note that the main program can call procedure one, but cannot call procedure twoa or twob (since they are nested two levels inside it). Procedure one can call procedure twoa or twob. Procedure twob can call procedure twoa or one. In Pascal, you cannot make a forward reference to a routine unless you declare the routine with the **forward** option (described in Chapter 5). If you used **forward** in this example, procedure twoa could call twob or one.

Chapter 3

Data Types

This chapter explains DOMAIN Pascal data objects. Here, we explain how you declare variables using the predeclared DOMAIN Pascal data types, and how you can define your own data types. In addition, this chapter shows how DOMAIN Pascal represents data types internally.

3.1 Data Type Overview

The data types of DOMAIN Pascal can be sorted into three groups -- simple, structured, and pointer. The following list shows the supported simple data types:

- INTEGERS -- DOMAIN Pascal supports the three predeclared integer data types **integer**, **integer16**, and **integer32**.
- REAL NUMBERS -- DOMAIN Pascal supports the three predeclared real number data types **real**, **single**, and **double**.
- BOOLEAN -- DOMAIN Pascal supports the predeclared data type **boolean**.
- CHARACTER -- DOMAIN Pascal supports the predeclared **char** data type.
- ENUMERATED -- DOMAIN Pascal supports enumerated data types.
- SUBRANGE -- DOMAIN Pascal supports a subrange of scalar data types. The scalar data types are integer, Boolean, character, and enumerated.

You can use the simple data types to build the following structured data types:

- SETS -- DOMAIN Pascal permits you to create a set of elements of a scalar data type.
- RECORDS -- DOMAIN Pascal supports the **record** and **packed record** data types.
- ARRAY -- DOMAIN Pascal supports the **array** data type. It also supports a predeclared character array type called **string**.

- FILES -- DOMAIN Pascal supports the **file** and **text** data types.

You can declare a pointer data type that points to any of the previous data types, or you can declare one of these other pointer data types:

- **Univ_ptr** -- DOMAIN Pascal supports a predeclared universal pointer data type that is compatible with any pointer type.
- PROCEDURE AND FUNCTION DATA TYPES -- DOMAIN Pascal supports a special data type that points to procedures and functions.

The program shown in Figure 3-1 contains sample declarations of all the data types. This program is available on-line and is named sample_types.

```
PROGRAM sample_types;

TYPE
   real_pointer     = ^real;      {This is a pointer type.     }
   writers = (Amy, David, Phil); {This is an enumerated type.}
   element = record              {This is a record type.     }
   atomic_number : INTEGER16;
   atomic_weight : SINGLE;
   half_life     : DOUBLE;
 end;

 VAR
    i1 : INTEGER;
    i2 : INTEGER16;
    i3 : INTEGER32;
    r1 : REAL;
    r2 : SINGLE;
    r3 : DOUBLE;
    consequences : BOOLEAN;
    onec : CHAR;
    teenage_years : 13..19;         {teenage_years is a subrange variable.}
    good_writers  : writers;       {good_writers is an enumerated variable.}
    tw           : SET OF writers;                   {tw is a set variable.}
    e            : element;                        {e is a record variable.}
    cat_nums : array[1..5] of INTEGER16;     {cat_nums is an array variable.}
    a_sentence : STRING; {a_sentence is an array variable of 80 characters.}
    hamlets_soliloquy : TEXT;    {hamlets_soliloquy is a text file variable.}
    periodic_table : FILE OF element;    {periodic_table is a file variable.}
    r1_ptr   : real_pointer;                   {r1_ptr is a pointer variable.}
    Any_Ptr  : UNIV_PTR;         {Any_ptr is a universal pointer variable.}
    pp       : ^PROCEDURE(IN x : INTEGER);
                                 {pp is a pointer to a procedure variable.}

   BEGIN
      writeln('Greetings.');
   END.
```

Figure 3-1. Program Declaring All Available Data Types

3.2 Integers

This section explains how to declare variables as integers, how to initialize integer variables, and how to define integer constants. It also explains how DOMAIN Pascal represents integers internally.

3.2.1 Declaring Integer Variables

DOMAIN Pascal supports the following three predeclared integer data types:

- **Integer** -- Use it to declare a signed 16-bit integer. A signed 16-bit integer variable can have any value from -32768 to +32767.
- **Integer16** -- Use it to declare a signed 16-bit integer. (**Integer** and **integer16** have identical meanings.)
- **Integer32** -- Use it to declare a signed 32-bit integer. A signed 32-bit integer variable can be any value from -2147483648 to +2147483647.

For example, consider the following integer declarations:

```
VAR
   x, y, z : INTEGER;
   quarts  : INTEGER16;
   social_security_number : INTEGER32;
```

If you want to define unsigned integers, you must use a subrange declaration (refer to the "Subrange" section later in this chapter).

3.2.2 Initializing Integer Variables -- Extension

DOMAIN Pascal permits you to initialize the values of integers within the variable declaration in most cases. You initialize a variable by placing a colon and equal sign (:=) immediately after the data type. For example, the following excerpt initializes X and Y to 0, and Z to 7000000:

```
VAR
   X,Y : INTEGER16 := 0;
   Z   : INTEGER32 := 7000000;
```

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
FUNCTION do_nothing(IN OUT x : INTEGER) : BOOLEAN;
VAR
   init_value : INTEGER := 0;          {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
   init_value : STATIC INTEGER := 0;
```

See Chapter 7 for more information on the **static** attribute.

3.2.3 Defining Integer Constants

When you declare an integer constant, DOMAIN Pascal internally represents the value as a 32-bit integer. For example, in the following declarations, DOMAIN Pascal represents both poco and grande as 32-bit integers.

```
CONST
   poco = 6;
   grande = 6000000;
```

You can specify an integer constant anywhere in the range -2147483648 to +2147483647.

It is also possible to compose constant integers as a mathematical expression. (Refer to the "Const Declaration Part" section in Chapter 2 for details.)

The predeclared integer constant **maxint** has the value +32767.

3.2.4 Internal Representation of Integers

DOMAIN Pascal represents a 16-bit integer (types **integer** and **integer16**) as two contiguous bytes, as shown in Figure 3-2. Bit 15 contains the most significant bit (MSB), and bit 0 contains the least significant bit (LSB). If the integer is signed, bit 15 contains the sign bit.

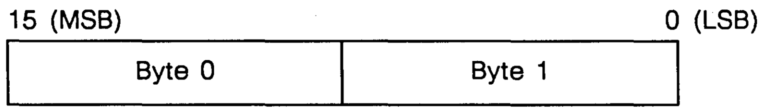

Figure 3-2. 16-Bit Integer Format

DOMAIN Pascal represents a 32-bit integer (type **integer32**) in four contiguous bytes as illustrated in Figure 3-3. The most significant bit in the integer is bit 31; the least significant bit is bit 0. If the integer is signed, bit 31 contains the sign bit.

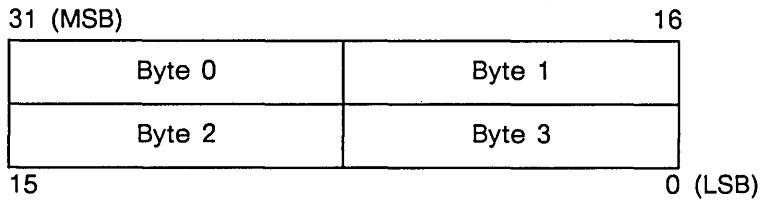

Figure 3-3. 32-Bit Integer Format

3.3 Real Numbers

This section describes how to declare variables as real numbers, how to define real numbers as constants, and how DOMAIN Pascal represents real numbers internally.

3.3.1 Declaring Real Variables

DOMAIN Pascal supports the following real data types:

- **Real** -- Use it to declare a signed single-precision real variable. DOMAIN Pascal represents a single-precision real number in 32 bits. A single-precision real variable has approximately seven significant digits.

- **Single** -- Same as **real.**

- **Double** -- Use it to declare a signed double-precision real variable. DOMAIN Pascal represents a double-precision real number in 64 bits. A double-precision real variable has approximately 16 significant digits.

For example, consider the following declarations:

```
VAR
   l, m, n : REAL;
   winning_time : SINGLE;
   cpu_time : DOUBLE;
```

3.3.2 Initializing Real Variables -- Extension

DOMAIN Pascal permits you to initialize the values of real numbers within the variable declaration in most cases. You initialize a value by placing a colon and equal sign (:=) immediately after the data type. For example, the following excerpt initializes variable pi to 3.14:

```
VAR
   pi   : SINGLE := 3.14;
```

If you declare a variable as **single** or **real**, and if you attempt to initialize it to a number with more than seven significant digits, then DOMAIN Pascal rounds (it does not truncate) the number to the first seven significant digits. For example, if you try to initialize pi this way

```
VAR
   pi   : SINGLE := 3.1415926535;
```

DOMAIN Pascal rounds pi to 3.141593.

As with integers, if the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
FUNCTION do_nothing(IN OUT x : REAL) : BOOLEAN;
VAR
   init_value : REAL := 0.0;          {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
   init_value : STATIC REAL := 0.0;
```

See Chapter 7 for information on the **static** attribute.

3.3.3 Defining Real Constants

When you use a real number as a constant, DOMAIN Pascal automatically defines the constant as a double-precision real number. This is true even if the constant can be accurately represented as a single-precision real number. However, when you use a real constant in a mathematical operation with a single-precision number, DOMAIN Pascal automatically rounds the constant to a single-precision number to produce a more accurate result. The following fragment defines four valid (and one invalid) real constants:

```
CONST
   N  = 24.57;     { Valid real number. }
   N2 = 2E19;      { Valid, symbolizes 2.0 * (10^19)      }
   G  = 6.67E-11;  { Valid, symbolizes 6.67 * (10^-11)    }
   X  = .5;        { Not a valid real number because it does }
                   { not contain a digit to the left of the  }
                   { decimal point. }
   X2 = 0.5;       { Valid real number.}
```

3.3.4 Internal Representation of Real Numbers

Single–precision floating–point numbers (types **real** and **single**) occupy four contiguous bytes of a long word, as shown in Figure 3–4. DOMAIN Pascal uses the IEEE standard format for representing 32–bit real values. Bit 31 is the sign bit with "1" denoting a negative number. The next eight bits contain the exponent plus 127. The remaining 23 bits contain the mantissa of the number without the leading 1. (DOMAIN Pascal stores the mantissa in magnitude form, not in two's–complement.)

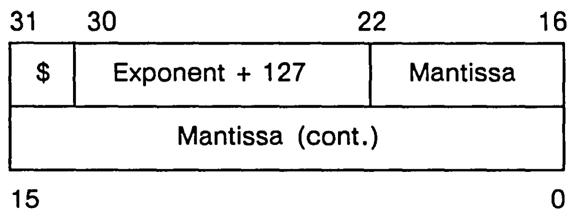

Figure 3–4. Single–Precision Floating–Point Format

For example, Pascal represents +100.5 in the following manner:

```
0100001011001001
0000000000000000
```

The number breaks into sign, exponent, and mantissa as follows:

```
sign                          -- 0 (positive)
exponent                      -- 10000101 (133 in decimal)
significant part of mantissa  -- 1001001
```

The exponent is 133; 133 is equal to 127 plus 6. Therefore, you can view the mantissa bits as follows:

```
bit 22 represents 2 to the fifth power
bit 21 represents 2 to the fourth power
bit 20 represents 2 to the third power
  .
  .
  .
bit 16 represents 2 to the negative first power.
```

You get 100.5 by adding ($2^6 + 2^5 + 2^2 + 2^{-1}$).

A number with a negative exponent is stored differently. Pascal represents 5E–2 as follows:

```
0011110101001100
1100110011001101
```

The number breaks into sign, exponent, and mantissa as follows:

```
sign                          -- 0 (positive)
exponent                      -- 01111010 (122 in decimal)
significant part of mantissa  -- 10011001100110011001101
```

The exponent is 122; 122 is equal to 127 plus -5. Therefore, you can view the mantissa bits as follows:

```
bit 22 represents 2 to the -6 power
bit 21 represents 2 to the -7 power
bit 20 represents 2 to the -8 power
.
.
.
bit 0 represents 2 to the -29 power
```

You get 5E-2 by adding $2^{-5} + 2^{-6} + 2^{-9}$ and so on.

DOMAIN Pascal represents double-precision floating-point numbers (type **double**) in eight bytes of a long word (64 bits). Figure 3-5 illustrates the format. The first bit (bit 63) contains the sign bit. The next 11 bits contain the exponent plus 1023. The remaining 52 bits hold the mantissa, without the leading 1.

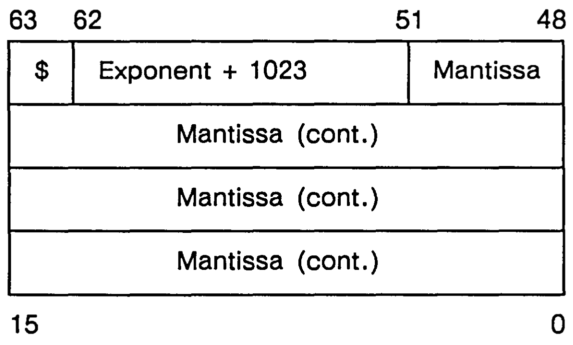

Figure 3-5. Double-Precision Floating-Point Format

3.4 Booleans

A Boolean variable can have only one of two values -- **true** or **false**. This section describes how you declare Boolean variables, how you define Boolean constants, and how DOMAIN Pascal represents Boolean variables internally.

3.4.1 Initializing Boolean Variables -- Extension

DOMAIN Pascal permits you to initialize the values of Boolean variables within the variable declaration in most cases. You initialize a value by placing a colon and equal sign (:=) immediately after the data type. For example, the following excerpt declares `liar` to be a Boolean variable with an initial value of `false`:

```
VAR
   liar : boolean := false;
```

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
FUNCTION do_nothing(IN OUT x : INTEGER) : BOOLEAN;
VAR
   liar : BOOLEAN := false;          {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
     liar : STATIC BOOLEAN := false;
```

See Chapter 7 for information on the **static** attribute.

3.4.2 Defining Boolean Constants

To define a Boolean constant, simply write the name of the constant, followed by an equal sign, and concluding with either **true** or **false**. For instance, the following excerpt defines constant `virtue` and sets it to **true**:

```
CONST
   virtue = true;
```

Notice that you do not enclose **true** or **false** inside a pair of apostrophes.

3.4.3 Internal Representation of Boolean Variables

DOMAIN Pascal represents Boolean values in one byte. The system sets all eight bits to 1 for **true** and sets all eight bits to 0 for **false**. However, a Boolean field in a packed record will have a different allocation (see the "Internal Representation of Packed Records" section later in this chapter for details).

3.5 Characters

This section describes how you declare a variable as a character data type, how you define characters as constants, and how DOMAIN Pascal represents characters internally.

3.5.1 Declaring Character Variables

Use the **char** type to declare a variable that holds one character; for example:

```
VAR
   a_letter, a_better_letter : CHAR;
```

To declare a variable that holds more than one character you must use an array or the predefined type **string** (both of which are detailed in the "Arrays" section later in this chapter).

3.5.2 Initializing Character Variables -- Extension

DOMAIN Pascal permits you to initialize the values of character variables within the variable declaration in most cases. You initialize a value by placing a colon and equal sign (:=) immediately after the data type. For example, the following excerpt declares `c1` as a **char** variable with an initial value of a:

```
VAR
   c1 : CHAR := 'a';
```

Notice that you must enclose the character in apostrophes.

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
FUNCTION do_nothing(IN OUT x : INTEGER) : BOOLEAN;
VAR
   best_grade : CHAR := 'A';          {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
      best_grade : STATIC CHAR := 'A';
```

See Chapter 7 for information on the **static** attribute.

3.5.3 Defining Character Constants

There are two common methods of assigning character constants. The first is to simply enclose a character inside a pair of apostrophes; for example

```
   CONST
      c1 = 'b';
```

This first method only works if the character is printable, but the second method works for all ASCII characters (printable or not). The second method uses the **chr** function (which is detailed in Chapter 4). As an example, suppose you want constant `bell` to contain the bell ringing character. The bell ringing character has an ASCII value of 7, so to assign this value to constant `bell` you can make the following declaration:

```
   CONST
      bell = CHR(7);
```

3.5.4 Internal Representation of Char Variables

DOMAIN Pascal stores the ASCII value of a **char** variable in one 8-bit byte.

3.6 Enumerated Data

An enumerated data type consists of an ordered group of identifiers. The only value you can assign to an enumerated variable is one of the identifiers. Here are declarations for four enumerated variables:

```
   VAR
      citrus         : (lemon, lime, orange, carambola, grapefruit);
      primary_colors : (red, yellow, blue);
      Beatles        : (John, Paul, George, Ringo);
      German_speaking_countries : (GDR, FRG, Switzerland, Austria);
```

In the code portion of your program, you can only assign the values `red, yellow,` or `blue` to variable `primary_colors`.

Notice that the elements of an enumerated type must be identifiers. Identifiers cannot begin with a digit, so, for example, the following declaration produces an "Improper enumerated constant syntax" error:

```
   VAR
      first_six_primes : (2, 3, 5, 7, 11, 13);   {error}
```

3.6.1 Internal Representation of Enumerated Variables

DOMAIN Pascal represents an enumerated variable in one 16-bit word. In this word, DOMAIN Pascal stores an integer corresponding to the ordinal position of the current value of the enumerated variable. For example, consider the following declaration:

```
   VAR
      pets : (cats, dogs, dolphins, gorillas, pythons);
```

Pets has five elements; DOMAIN Pascal represents those five elements as integers from 0 to 4 as shown in Table 3-1.

Table 3-1. Representation of an Enumerated Variable

`pets := cats`	0000000000000000
`pets := dogs`	0000000000000001
`pets := dolphins`	0000000000000010
`pets := gorillas`	0000000000000011
`pets := pythons`	0000000000000100

3.7 Subrange Data

A variable with the subrange type has a valid range of values that is a subset of the range of another type called the base type. When you define a subrange, you specify the lowest and highest possible value of the base type. You can specify a subrange of integers, characters, or any previously-defined enumerated type. The following fragment declares four different subrange variables:

```
TYPE
    mountains = (Wachusett, Greylock, Washington, Blanc, Everest);
    {Mountains is an enumerated type.}

VAR  {The following four variables all have subrange types.}
   teenage_years      : 13..19;                 {Subrange of INTEGER.}
   positive_integers  : 1..MAXINT;              {Subrange of INTEGER.}
   capital_letters    : 'A'..'Z';               {Subrange of CHAR.}
   small_mountains    : Wachusett..Washington;  {Subrange of MOUNTAINS.}
```

Currently, DOMAIN Pascal does not support subrange checking. For example, if you try to assign the value 25 to teenage_years, DOMAIN Pascal does not report an error. However, you can use the **in_range** function to determine whether 25 is within the declared subrange. (See Chapter 4 for information on the **in_range** function.)

3.7.1 Internal Representation of Subranges

The storage allocation for subrange variables is the same as that for their base types. However, a subrange field in a packed record will have a different allocation (see the "Internal Representation of Packed Records" section later in this chapter for details).

3.8 Sets

A set in DOMAIN Pascal is virtually identical to a set in standard mathematics. For instance, DOMAIN Pascal can compute unions and intersections of DOMAIN Pascal set variables just like you can find unions and intersections of two mathematical sets. Refer to the "Set Operations" listing in Chapter 4 for information on using sets in the action part of your program.

3.8.1 Declaring Set Variables

The format for specifying a set variable is as follows:

set of **boolean** | **char** | enumerated_type | subrange_type

For example, consider the following set declarations:

```
TYPE
   very    = (ochen, sehr, tres, muy);
   lowints = 0..100;

VAR
   ASCII_values : set of char;          {Char is base type.}
   possibilities : set of boolean;      {Boolean is base type.}
   capital_letters : set of 'A'..'Z';  {Subrange of CHAR is base type.}
   lots : set of very;                  {Enumerated type is base type.}
   digits : set of lowints;             {lowints is base type.}
```

If the base type is a subrange of integers, then the low end of the subrange cannot be a negative number. Also, the high end of the subrange cannot exceed 255.

If the base type is an enumerated type, the enumerated type cannot contain more than 2048 elements.

NOTE: DOMAIN Pascal lets you declare **packed** set variables or **packed** set types. However, the **packed** designation does not affect the amount of memory the compiler uses to represent the set; that is, the compiler ignores the designation. Furthermore, specifying a **packed set** triggers the following warning message:

```
Warning:  SET cannot be PACKED.
```

3.8.2 Initializing Set Variables -- Extension

In most cases, you can initialize set variables with an assignment statement in the variable declaration. For example, consider the following set variable initializations:

```
TYPE
   unstable_elements = (U, Pl, Ei, Ra, Xe);

VAR
   letters : set of CHAR := ['A', 'E', 'I', 'O', 'U'];
   humanmade_elements : set of unstable_elements := [Pl, Ei];
```

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
FUNCTION assign_grades(IN OUT score : INTEGER) : BOOLEAN;
VAR
   grades : set of CHAR := ['A', 'B', 'C', 'D', 'E'];             {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
   grades : STATIC set of CHAR := ['A', 'B', 'C', 'D', 'E'];
```

See Chapter 7 for information on the **static** attribute.

Refer to the "Set Operations" listing in Chapter 4 for more information on set assignment.

3.8.3 Internal Representation of Sets

A set can contain up to 256 elements; their ordinal values are 0 to 255. Sets are stored as bit masks, with one bit representing one element of the set. The number of bits that DOMAIN Pascal allocates to a set is the number of elements in the set, rounded up to a multiple of 16 bits. That is, a set occupies the minimum number of words that provides one bit per element. Consequently, the minimum storage size for a set is one word (16 bits) and the maximum size is 16 words (256 bits).

For example, suppose you define an enumerated type named `Greek_letters`, with values `Alpha`, `Beta`, `Gamma`, and so forth, up to `Omega`. You can then declare a set of `Greek_letters` as follows:

```
VAR
   Greek_alphabet : SET of Greek_letters
```

`Greek_alphabet` has 24 values, and therefore, `Greek_alphabet` requires at least 24 bits. The nearest word boundary is 32 bits, so DOMAIN Pascal allocates 32 bits (2 words) for the variable. It then stores the values this way:

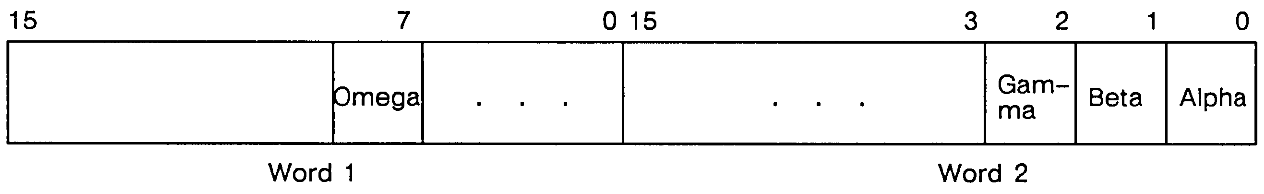

Figure 3-6. Storage of Sample Set

If the base type of the set is a subrange of integers or a subrange of char, then the ordinal value of the high end of the subrange determines the amount of space required to store the set. For example, consider the following two set declarations:

```
TYPE
   possible_values = 80..170;
   small_letters   = 'a'..'z';

VAR
   pos : set of possible_values;
   sma : set of small_letters;
```

DOMAIN Pascal stores variable `pos` in 11 words (176 bits). That is because the highest ordinal value of the base type (`possible_values`) is 170. The next word boundary up from 170 is 176.

DOMAIN Pascal stores variable `sma` in eight words (128 bits). In the base type (`small_letters`), the ordinal value of `z` is 122. The next word boundary up from from 122 is 128.

3.9 Records

A record variable is composed of one or more different components (called fields) which may have different types. DOMAIN Pascal supports the two standard kinds of records: fixed records and variant records. The following subsections describe both kinds.

3.9.1 Fixed Records

A fixed record consists of zero or more fields. Each field can have any valid DOMAIN Pascal data type. To declare a fixed record type, issue a declaration of the following format:

```
type
      record_name = record
                    field1;
           .          .
           .          .
           .          .
                    fieldN;
      end;
```

Each *field* has the following format:

```
field_name1, ... field_nameN : datatype;
```

For example, consider the following three record declarations:

```
TYPE
   student = record  {Contains two fields.}
             name  : array[1..30] of char;
             id    : INTEGER16;
   end;

   element = record  {Contains four fields.}
             name           : array[1..15] of char;
             symbol         : array[1..2] of char;
             atomic_number  : 1..120;
             atomic_weight  : real;
   end;

   weather = record  {Contains five fields.}
             station         : array[1..3] of char;
             sky_condition   : (fair, ptly_cloudly, cloudy);
             windspeed       : 1..100;
             winddirection   : 1..360;
             pressure        : single;
   end;

VAR
   new_students : student;
   noble_gases  : element;
   w1           : weather;
```

Note that you can declare a record type as the data type of a field. For example, notice the changes in the declaration of the record weather:

```
TYPE
   wind = record
            speed : 1..100;
            direction : 1..360;
   end;

   weather = record
               station        : array[1..3] of char;
               sky_condition  : (fair, ptly_cloudly, cloudy);
               gradient       : wind;
               pressure       : single;
   end;
```

NOTE: A common mistake is to misuse the equal sign (=) and the colon (:). When declaring a record in the **type** declaration part, put an equal sign between the record_name and the keyword **record**. When declaring a record in the **var** declaration part, put a colon between them. When specifying a field (whether in a **type** declaration part or a **var** declaration part), put a colon between the field_name and its data type.

3.9.2 Variant Records

A variant record is a record with multiple data type possibilities. When you declare a variant record, you specify all the possible data types that the record can have. You also specify the condition for selecting among the multiple possibilities.

Here is another way of viewing the distinction between fixed records and variant records. The system views a fixed record variable at runtime as having the same group of data types from one use of the variable to another. However, the system views a variant record variable at runtime as having a flexible group of data types from one use of the variable to another.

The variant record has the following format:

type
    record_name = **record**
        fixed_part;
        variant_part;
    end;

The *fixed_part* of a variant record is optional. It looks just like a fixed record. In other words, the *fixed_part* consists of one or more fields each having the following format:

field_name1, ... *field_nameN* : datatype;

The variant_part of a variant record takes the following format:

case *tag_field* data_type **of**
    constantlist1 : (field; ... *fieldN*);
        .
        .
        .
    constantlistN : (field; ... *fieldN*);

The constantlist is one or more constants that share the same data_type. For instance, if data_type is **integer**, then every constant in constantlist must be an integer. You associate one or more fields with each constantlist. With one exception, each field has the same syntax as a field in the fixed part. The one exception is that *fieldN* can itself be a variant_part.

Note that you can optionally associate a *tag_field* with the data_type. The *tag_field* is simply an identifier followed by a colon (:). You can use the *tag_field* to select the desired variant at runtime. For more information on *tag_fields*, see the "Record Operations" listing in Chapter 4.

Consider the following declaration for variant record type worker. Worker contains a fixed part and a variant part. The fixed part contains two fields (employee and id_number). The data_type of the variant part is worker_groups, which is an enumerated type. Wo has two possible values, exempt and non_exempt. When wo is exempt, the field name is yearly_salary which is an **integer32** data type, and when wo is non_exempt, the field name is hourly_wage which has a real data type.

```
TYPE
   worker_groups = (exempt, non_exempt);     {enumerated type}
   worker = record     {record type}
            employee : array[1..30] of char; {field in fixed part}
            id_number : integer16;           {field in fixed part}
    CASE wo : worker_groups OF        {variant part}
              exempt : (yearly_salary   : integer32);
              non_exempt : (hourly_wage : real);
    end;
```

My_code is a variant record that does not contain a fixed part. The data type of the variant part is **integer**, so the **case** portion declares integer constants. Choosing 1, 2, 3, and 4 as the constants is totally arbitrary; you could pick any four integers. These constants serve no purpose except to establish the fact that there are four choices. The fields themselves provide four different ways to view the same 4-byte section of main memory.

```
    my_code = record
    CASE integer OF          {variant part}
              1 : (all : array[1..4] of char);
              2 : (first_half : array[1..2] of char;
                   second_half : array[1..2] of char);
              3 : (x1  : integer16;
                   x2  : boolean;
                   x3  : char);
              4 : (rall : single);
    end;
```

NOTE: The preceding example shows four parts that take up exactly four bytes. However, it is perfectly valid to declare parts that take up differing numbers of bytes.

3.9.3 Unpacked Records and Packed Records

DOMAIN Pascal supports regular (unpacked) records and "packed" records. You declare a packed record by putting the keyword **packed** prior to **record** in the record declaration; for example:

```
VAR
   student : PACKED record
             ages       : 10..20;
             grade      : (seventh, eighth, ninth, tenth, eleventh, twelfth);
             graduating : boolean;
   end;
```

The advantage to declaring a packed record is that it can save space. The disadvantage is that you cannot pass a field from a packed record as an argument to a procedure (including predeclared procedures like **read**). The next subsection details the space savings of packing. Note that you should not directly manipulate fields in a packed record. If you want to perform some operation that changes the value of an existing field in a packed record, use the following steps:

1. Assign the value of the field to a variable of the same type.
2. Perform the operation on the variable.
3. Assign the value of the variable to the field of the packed record.

3.9.4 Initializing Data in a Record -- Extension

DOMAIN Pascal permits you to initialize a record in the variable declaration portion of the program unless that declaration comes within a procedure or function and the record has not been declared **static.** (See Chapter 7 for information on the **static** attribute.) You can initialize some or all of the fields in a record.

To initialize a field in a record, enter a declaration with the following format

```
var
      name_of_record_variable : type_of_record :=
                  [ init,
                      . ,
                      . ,
                      . ,
                    init ];
```

where init is a statement having one of the following formats:

```
field_name := initial_value
```

or

```
initial_value
```

If you choose the second format, DOMAIN Pascal assumes that the initial_value applies to the next field_name in the record definition. For example, consider the following record initialization:

```
TYPE
   messy = record
           rx  : real;
           c   : char;
           abc : array[1..3] of integer;
           case integer of
              0 : (i32 : integer32);
              1 : (i16 : integer);
              2 : (hb, lb : char);
           end;

VAR
   very : MESSY :=
          [ c := 'X',
            [-1, -2, -3],
            rx := 123.456,
            'Y',
            lb := 'a',
            hb := 'z' ];
```

The preceding example initializes field c to 'X'. The next declaration [-1, -2, -3] applies to field abc (because it follows field c). Field rx gets initialized to 123.456. Then, field c gets reinitialized to 'Y' (because it follows field rx). Finally, the third field in the variant portion of the record gets initialized, with field lb getting the value 'a' and field hb getting set to 'z'.

3.9.5 Internal Representation of Unpacked Records

In a record, DOMAIN Pascal allocates the same amount of space for each field that it would have required if it were not part of the record. The compiler aligns fields of Boolean, character, and character array types on byte boundaries. It aligns fields of other types on word boundaries. The compiler aligns the record itself on a word boundary.

For example, consider the following unpacked record type definition:

```
VAR
   Idx_rec = record
               entry : integer16;
               new   : boolean;
               next  : ^idx_rec;
   end;
```

Figure 3-7 shows how DOMAIN Pascal stores variables of this type.

In this figure, the first 16 bits of the structure contain the value of the first field, entry. The next eight bits holds the Boolean value of new, followed by one unused byte. The 32-bit address field, next, starts on the next word boundary.

Because of alignment requirements, the order in which you declare fields of the record can have a substantial impact upon the actual amount of memory required to store the record. In general, to minimize memory requirements, declare those fields that have the same size to be adjacent to each other.

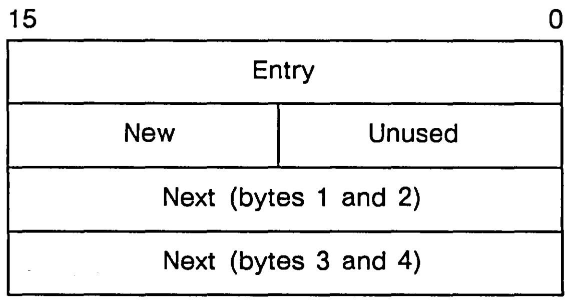

Figure 3-7. Sample Record Layout

3.9.6 Internal Representation of Packed Records

Table 3–2 shows the space required for fields in packed records.

Table 3–2. Storage of Packed Record Fields

Data Type of Field	Space Allocation
Integer, Integer16	16 bits; word–aligned.
Integer32, Real, Single	32 bits; word–aligned.
Double	64 bits; word–aligned.
Boolean	1 bit; bit–aligned.
Char	8 bits; byte–aligned.
Enumerated	Number of bits required for largest ordinal value; bit–aligned.
Subrange	Subrange of Char fields require eight bits; all other subrange fields take up the number of bits required for their extreme values. Subrange of Char fields are byte–aligned. All other subrange fields are bit–aligned.
Set	If fewer than 32 elements, then exactly one bit per element; if more than 32 elements, then same size as unpacked set. Bit–aligned.
Array	*Never* packed; requires the same space as an array outside of a packed record. (See the "Internal Representation of Arrays" section.)
Pointer	32 bits.

DOMAIN Pascal always starts the first field of a packed record on a word boundary. After the first field, if the exact number of bits required for the next field crosses zero or one 16–bit boundary, the field starts in the next free bit. If the field would cross two or more 16–bit boundaries, it starts at the next 16–bit boundary. Pascal allocates fields left to right within bytes and then by increasing byte address.

The minimum size of a packed record is 16 bits.

In packed records, characters are byte–aligned. Structured types, except for sets, are aligned on word boundaries. Sets are aligned only if they cross two or more 16–bit boundaries.

The following type declaration, along with Figure 3-7, illustrates the storage of a packed record type.

```
TYPE
   Shapes = (Sphere, Cube, Ovoid, Cylinder, Tetrahedron);
   Uses = (Toy, Tool, Weapon, Food);
   Characteristics = PACKED RECORD
                     Mass     : Real;
                     Shape    : Shapes;
                     B        : Boolean;
                     Purpose  : SET OF Uses;
                     Low_temp : -100..40;
                     Class    : 'A'..'Z';
   end;
```

The fields require the following number of bits:

```
Mass                32 bits  (word-aligned)
Shape                3 bits  (bit-aligned)
B                    1 bit   (bit-aligned)
Purpose              4 bits  (bit-aligned)
Low_temp             8 bits  (bit-aligned)
Class                8 bits  (word-aligned)
```

(The variable low_temp requires eight bits because it can take a range of 140 values (-100 to +40) and seven bits can represent only 128 values.)

DOMAIN Pascal represents fields in the same order you declared them, as shown in Figure 3-8.

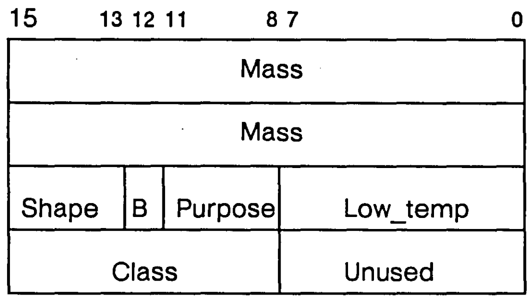

Figure 3-8. Sample Packed Record

In this example, the order of field declaration has been chosen very carefully. The whole record takes up only eight bytes, and out of the eight bytes, only eight bits are unused. If the fields had been declared in a different order, the record might have taken up 10 or 12 bytes.

3.10 Arrays

Like standard Pascal, DOMAIN Pascal supports array types. An array consists of a fixed number of elements of the same data type. This data type is called the component type. The component type can be any predeclared or user-declared data type.

You specify the number of elements the array contains through an index type. The index type must be a subrange expression. DOMAIN Pascal permits arrays of up to seven dimensions. You specify one subrange expression for each dimension.

This fragment includes declarations for five arrays:

```
TYPE
   {elements is an enumerated type.}
   elements       = (H, He, Li, Be, B, C, N, O, Fl, Ne);

VAR
   {Here are the five array declarations.}
   test_data      : array[1..100] of INTEGER16;
   atomic_weights : array[H..Be] of REAL;         {Range defined in TYPE}
                                                  {declaration.        }
   last_name      : array[1..15] of CHAR;
   a_thought      : STRING;
   lie_test       : array[1..4, 1..2] of BOOLEAN; {2-dimensional array. }
```

Notice that variable a_thought is of type **string. String** is a predefined DOMAIN Pascal array type. DOMAIN Pascal defines **string** as follows:

```
TYPE
   string = array[1..80] of CHAR;
```

In other words, **string** is a data type of 80 characters. Use **string** to handle lines of text conveniently.

See the "Array Operations" listing in Chapter 4 for a description of array bound checking.

3.10.1 Initializing Variable Arrays -- Extension

DOMAIN Pascal permits you to initialize the components in an array within the variable declaration statement unless that declaration is in a function or procedure and the array has not been declared **static.** (See Chapter 7 for more information on the **static** attribute.) DOMAIN Pascal initializes only those components for which it finds initialization data. It does not initialize the other components. For example, if an integer array consists of 10 components and you specify six initialization constants, then DOMAIN Pascal initializes the first six components and leaves the remaining four components uninitialized.

The method you use to initialize an array depends on the type of the array.

If the component type of the array is **char,** then you specify an assignment operator (:=) followed by a string. For example, consider the following initializations:

```
CONST
   msg1 = 'This is message 1';

VAR
   s1          : array[1..40] of CHAR := 'Quoted strings are ok';
   s2          : array[1..30] of CHAR := msg1;
   blank_line  : STRING               := chr(10);  {New line}
```

If the component type is something other than **char,** you must initialize the components of the array individually. (If the component type is **char,** you can initialize the components individually, but it's usually easier to do it as a string.) You do this by specifying an assignment operator (:=) followed by the values of the components in the array. You must enclose these values inside a pair of brackets and separate each value with commas. For example, consider the following array initializations:

```
VAR
   I : array[1..6] of INTEGER16 := [1, 2, 4, 8, 16, 32];
   R : array[1..3] of SINGLE := [-5.2, -7.3, -2E-3];
   B : array[1..5] of BOOLEAN := [true, false, true, true, true];
```

The following fragment demonstrates how to provide initialization data for two 2-dimensional arrays (I2 and B2). I2 has two subrange index types (1..2 and 1..6). The first index type comprises two values, so you must supply two rows of brackets. The second scalar is 1..6, so you must specify six values for each row.

```
VAR
   I2 : array[1..2, 1..6] of INTEGER16 := [
                                            [1, 2, 3, 4, 5, 6],
                                            [7, 5, 9, 1, 2, 8],
                                           ];

   B2 : array[1..4, 1..3] of BOOLEAN := [
                                          [true, true,  true],
                                          [true, false, false],
                                          [false, true, false],
                                          [false, false, false],
                                         ];
```

3.10.1.1 Defaulting the Size of An Array -- Extension

When you initialize an array in the **var** declaration part of a program, you can let DOMAIN Pascal determine the size of the array for you. To accomplish this, put an asterisk (*) in place of the upper bound of the array declaration. For example, in the following fragment, the upper bound of `init4` is 18; the upper bound of `init5` is 22; and the upper bound of `init6` is 4. The compiler defines the upper bound once it has counted the number of initializers.

```
CONST
   msg5 = 'And this is message 5.';

VAR
   init4 : ARRAY[1..*] of CHAR := 'This is message 4.';
   init5 : ARRAY[1..*] of CHAR := msg5;
   init6 : ARRAY[1..*] of INTEGER16 := [1, -17, 35, 46];
```

NOTE: You can only use an asterisk in the index type if you supply an initialization value for the array. For example, the following fragment causes a "Size of TABLE1 is zero" warning:

```
VAR
   table1 : array[1..*] of integer16;
```

3.10.1.2 Using Repeat Counts to Initialize Arrays -- Extension

It is quite tedious to individually initialize every component in a large array. However, DOMAIN Pascal provides a *repeat count* feature that simplifies initialization.

There are two forms of repeat counts. The first form takes the following syntax:

n **of** constant

This form tells the compiler to initialize n components of the array to the value of constant. N can be an integer or an expression that evaluates to an integer. The following initializations demonstrate this form of the repeat count:

```
CONST
   x = 50;
VAR
   a : array[1..1024] of INTEGER16 := [512 OF 0, 512 OF -1];
   b : array[1..400] of REAL := [x of 3.14, 400-x OF 2.7];
```

In the preceding example, DOMAIN Pascal initializes the first 512 values of array a to 0 and the second 512 values to -1. DOMAIN Pascal also initializes the first 50 components of array b to 3.14 and the remaining 350 components to 2.7.

The second form of the repeat count takes the following syntax:

```
* of constant
```

The asterisk (*) tells DOMAIN Pascal to initialize the remainder of the components in the array to the value of constant. The following initializations demonstrate this form of the repeat count:

```
VAR
   c : array[1..2000] of INTEGER16 := [* of 0];
   d : array[1..50] of BOOLEAN := [12 of true, * of false];
```

In the preceding example, DOMAIN Pascal initializes all 2000 components in array c to 0. DOMAIN Pascal also initializes the first 12 components of array d to true and the remaining 38 components to false.

You can use repeat counts to initialize multidimensional arrays; however, you must initialize a multidimensional array column by column rather than all at once. For example, compare the right and wrong ways to initialize a 2-dimensional array:

```
VAR
   x : array[1..2, 1..5] of INTEGER16 := [10 of 0];   {wrong}
   y : array[1..2, 1..5] of INTEGER16 := [* of 0];    {wrong}
   z : array[1..2, 1..5] of INTEGER16 := [
                                         [5 of 0],
                                         [5 of 0],
                                        ];            {right}
   q : array[1..2, 1..5] of INTEGER16 := [
                                         [* of 0],
                                         [* of 0],
                                        ];            {right}
```

3.10.2 Internal Representation of Arrays

With two exceptions, the total amount of memory required to store an array equals the number of elements in the array times the amount of space required to store one element. The amount of space for one element depends on the component type of the array, as shown in Table 3-3.

The two exceptions to this rule are arrays of **booleans** and **chars**. If the component type of the array is either **boolean** or **char**, the storage space for an array declaring an odd number of elements is represented in an even number of bytes. For example, if you declare **boolean** array b as

```
VAR
   b  : array[1..5] of boolean;
```

DOMAIN Pascal reserves six bytes of memory for b.

Table 3-3. Size of One Element of an Array

Base Data Type	Size of One Element
Integer16 or Integer	16 bits
Integer32	32 bits
Single or Real	32 bits
Double	64 bits
Boolean	8 bits
Char	8 bits
Subrange	size of base type of subrange
Enumerated	16 bits

Given a 2-dimensional array of the following declaration:

```
a : array[1..2, 1..3] of integer16;
```

DOMAIN Pascal represents it in the following order:

```
a[1,1] first
a[1,2] second
a[1,3] third
a[2,1] fourth
a[2,2] fifth
a[2,3] sixth
```

3.11 Files

When you open a file for I/O access, you must specify a file variable that will be the pseudonym for the actual pathname of the file. Thereafter, you specify the file variable (not the pathname) to refer to the file. DOMAIN Pascal supports the **file** data type and the **text** data type. (Throughout this manual, the word "**file**," in boldface type, means the **file** data type, and the word "file," in roman type, means a disk file.)

A variable with the **text** type specifies a DOMAIN file with the UASC (unstructured ASCII) attribute. For example, the following declaration establishes variable f1 as a synonym for a UASC file:

```
VAR
   f1 : text;
```

UASC files contain sequences of ASCII characters representing variable-length lines of text. You can read or write entire lines of a UASC file. You can read from or write to a UASC file the values of a variable of any type. Chapter 8 describes UASC files in more detail.

You specify a **file** variable with the following format:

variable : **file of** base_type;

A variable with the **file** type specifies a file composed of values having the base_type. That is, the only permissible values in such a file all have the same data type, that of the base_type. The base_type can be any type except a pointer, **file**, or **text** type. The **file** variable type creates a DOMAIN record–structured file whose record size is the size of the data type. Chapter 8 describes record–structured (rec) files in more detail. For example, the following declaration creates a file type corresponding to a file that consists entirely of student records:

```
TYPE
   student = record
             name : array[1..30] of char;
             id   : integer32;
   end;
   U_of_M  : FILE OF student;
```

The DOMAIN operating system stores each occurrence of `student` in 38 bytes: 30 bytes for name, 4 bytes for id, and 4 bytes (system supplied) for a record count field.

If you redefine the **text** data type (for example, in a **type** statement), it loses its UASC attribute. For example, if you specify the following declaration:

```
TYPE
   text = FILE OF char;
```

then **text** is no longer a UASC type. It is a record file with a record size of one byte.

For more information on DOMAIN file types, see *Programming With General System Calls.*

3.12 Pointers

A pointer variable points to a dynamic variable. In DOMAIN Pascal, the value of a pointer variable is a variable's virtual address. DOMAIN Pascal supports the pointer type declaration of standard Pascal as well as a special **univ_ptr** data type and procedure and function pointer types. This section details the declaration of pointer types. You should also refer to the "Pointer Operations" listing of Chapter 4 for information on using pointers in your programs.

3.12.1 Standard Pointer Type

To declare a pointer type, use the following format:

```
type
      name_of_type = ^typename
```

You can specify any data type for typename. The pointer type can point only to variables of the given **type**. For example, consider the following pointer type and variable declarations:

```
TYPE
   ptr_to_int16 = ^integer16;  {Points only to integer16 variables.}
   ptr_to_real  = ^real;       {Points only to real variables.}
   studentptr = ^student;      {Points only to student record variables.}
   student = record
             name : array[1..25] of char;
             id   : integer;
             next_student : studentptr;
   end;

VAR
   x   : integer16;
   p_x  : ptr_to_int16;
   half_life : real;
   p_half_life : ptr_to_real;
   Brown_Univ : student;
```

3.12.2 Univ_ptr -- Extension

The predeclared data type **univ_ptr** is a universal pointer type. A variable of type **univ_ptr** can hold a pointer to a variable of any type. You can use a **univ_ptr** variable in the following contexts only:

- Comparison with a pointer of any type
- Assignment to or from a pointer of any type
- Formal or actual parameter for any pointer type
- Assignment to the result of a function

Note that you *cannot* de-reference a **univ_ptr** variable. De-referencing means finding the contents at the logical address that the pointer points to. You must use a variable of an explicit pointer type for the de-reference. Please see the "Pointer Operations" listing in Chapter 4 for more information on **univ_ptr**.

3.12.3 Procedure and Function Pointer Data Types -- Extension

DOMAIN Pascal supports a special pointer data type that points to a procedure or a function. By using procedure and function data types, you can pass the addresses of routines obtained with the **addr** predeclared function. (See the **addr** listing of Chapter 4 for a description of this function.) You may only obtain the addresses of top-level procedures and functions; you cannot obtain the addresses of nested or explicitly declared **internal** procedures and functions. (See Chapter 5 for details about **internal** procedures.)

Procedure and function pointer type declarations are the same as regular procedure and function declarations, except for the following:

- The procedure or function has no identifier; in other words, the procedure or function does not have a name.
- The type declaration begins with an up-arrow (just like standard pointer types).

For example, consider the following variable declarations:

```
VAR
   i,j : INTEGER;
   func_ptr : ^FUNCTION(a : char) : INTEGER32;
   proc_ptr : ^PROCEDURE(x,y,z  : real;
                      quarts : integer16);
```

3.12.4 Initializing Pointer Variables -- Extension

DOMAIN Pascal permits you to initialize the values of pointer variables within its variable declaration in most cases. You initialize a value by placing a colon and equal sign (:=) immediately after the data type. For example, the following fragment declares `my_ptr` as a type `ptr_to_int16` with an initial value of `NIL`:

```
TYPE
   ptr_to_int16 = ^integer16;
VAR
   my_ptr : ptr_to_int16 := NIL;
```

If the variable declaration occurs within a procedure or function, you *cannot* initialize the variable at the declaration unless it has been declared **static**. This is because storage within routines is dynamic and so variables in them do not necessarily retain their values between executions. For example, the following is incorrect:

```
TYPE
   ptr_to_int16 = ^integer16;

FUNCTION do_nothing(IN OUT x : INTEGER) : BOOLEAN;
VAR
   my_ptr : ptr_to_int16 := NIL;              {Wrong!}
```

This is the correct way to initialize the variable at its declaration in a routine:

```
   my_ptr : STATIC ptr_to_int16 := NIL;
```

See Chapter 7 for information on the **static** attribute.

3.12.5 Internal Representation of Pointers

DOMAIN Pascal stores pointer variables in the 32-bit structure shown in Figure 3-9.

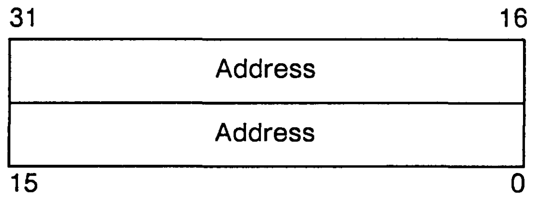

Figure 3-9. Pointer Variable Format

A pointer to a procedure or function (a DOMAIN Pascal extension) points to the starting address of that routine.

3.13 Putting Variables Into Sections -- Extension

A "section" is a named area of code or data. At runtime, the code or data in a particular section occupies contiguous logical addresses. By default, all variables that you declare in a **var** declaration part are stored in the DATA$ section. However, DOMAIN Pascal lets you assign variables to sections other than DATA$. Named data sections are synonymous with named common blocks in FORTRAN.

To specify a data section, place the section name (any valid identifier) after the reserved word **var**. You must enclose the section name inside parentheses. That is, use the following format to declare a section name for a **var** declaration part.

```
var (section_name)
      identifier_list1 : data_type1;
            .          .
            .          .
            .          .
      identifier_listN : data_typeN;
```

All the variables named in all the identifier_lists will be stored in section_name. Since you can put multiple **var** declaration parts in the same program, you can create multiple named sections. If you do not specify a section_name, DOMAIN Pascal puts the variables in the DATA$ section.

DOMAIN Pascal allocates variables defined in a **var** declaration part sequentially within the specified section. If more than one **var** declaration specifies the same section name, the subsequent declarations are considered to be continuations of the first declaration.

By forcing certain variables into the same section, you can reduce the number of page faults and thus make your program execute faster. For example, suppose you declare the following three variables:

```
VAR
   x      : integer16;
   b_data : array[1..5000] of integer16;
   y      : single;
```

Further suppose that whenever you need the value of x, you also need the value of y. By default, DOMAIN Pascal places x, b_data, and y inside the DATA$ section. The DATA$ section encompasses 10 pages (1 page = 1024 bytes). There is no way to ensure that x and y will be on the same page in DATA$ because DOMAIN Pascal might place b_data in between x and y. However, by putting x and y in the same named section, you can improve the odds to over 99%. For example, to put x and y into section important, you must issue the following declarations:

```
VAR (important)
   x : INTEGER16;       {will go into section "important"}
   y : REAL;            {will go into section "important"}

VAR
   b_data  :  array[1..5000] of INTEGER16; {will go into section "DATA$"}
```

Sections are very important at bind time. For complete information on the DOMAIN binder, see the *DOMAIN Binder and Librarian Reference*.

3.14 Attributes for Variables and Types -- Extension

DOMAIN Pascal supports attributes for variables and types. These attributes supply additional information to the compiler when you declare a variable or a type.

DOMAIN Pascal currently supports three of these attributes: **volatile**, **device**, and **address**. The **volatile** and **device** attributes enable you to turn off certain optimizations that would otherwise ruin programs that access device registers or shared memory locations. The **address** attribute associates a variable with a specific virtual address.

You specify these attributes immediately prior to the type; that is, immediately after the colon or equal sign. You must place them inside a pair of brackets; for example:

```
TYPE
   int_array = [VOLATILE] array[1..10] of integer;
VAR
   x : [DEVICE] integer16;
```

To specify more than one attribute for a particular data type or variable, separate the attributes with commas.

The following subsection detail the attributes.

3.14.1 Volatile -- Extension

Volatile informs the compiler that memory contents may change in a way that the compiler cannot predict. There are two situations, in particular, where this might occur:

- The variable is in a shared memory location accessed by two or more processes.

- The variable is accessible through two different access paths. (That is, multiple pointers with different base types refer to the same memory locations.)

In both of these situations, it is crucial that you tell the compiler *not* to perform certain default optimizations.

For example, the following module causes optimizations leading to erroneous code:

```
Module volatile_example;

VAR
   p : ^integer;

Procedure Init(VAR v : integer);
BEGIN
   p := addr(v);
END;

Procedure Update;
BEGIN
   p^ := p^ + 1; {anonymous path.}
END;

Procedure Top;
VAR
   i : integer;
BEGIN
   Init(i);
   i := 0;              {Visible modification.    }
   while i < 10 do      {Visible reference.       }
      update;           {Hidden modification to i.}
END;
```

However, you can prevent these destructive optimizations if you change the declaration of variable i to:

```
VAR
   i : [volatile] integer;
```

3.14.2 Device -- Extension

Device informs the compiler that a device register (control or data) is mapped to a specific virtual address. **Device** prevents the same optimizations that **volatile** prevents, and it also prevents two other optimizations.

By default, the compiler optimizes certain adjacent references by merging them into one large reference. The **device** attribute prevents this optimization.

For example, consider the following fragment:

```
VAR
   a,b : integer16;
BEGIN
   a := 0;
   b := 0;
```

By default, the compiler optimizes the two 16-bit assignments by merging them into one 32-bit assignment. (That is, at runtime, the system assigns a 32-bit zero instead of assigning two 16-bit zeros.) By specifying the **device** attribute, you suppress this optimization.

The **device** attribute also prevents the compiler from generating gratuitous read–modify–write references for device registers. That is, specifying a variable as **device** causes the compiler to avoid using instructions that do unnecessary reads.

Now, consider an example. Suppose kb in the following fragment is a device register that accepts characters from the keyboard.

```
TYPE
   keyboard = char;

VAR
   c, c1 : char;
   kb    : ^keyboard;

BEGIN
   c  := kb^;
   c1 := kb^;
```

The purpose of the program is to read a character from the keyboard and store it in c, then read the next character and store it in c1. However, the compiler, unaware that the value of kb can be changed outside of the block, optimizes the code as follows. It stores the value of kb in a register, and thus assigns both c and c1 identical values. Obviously, this is not what the programmer intended since DOMAIN Pascal assigns the same character to both c and c1. To ensure that DOMAIN Pascal reads kb twice, declare it as:

```
TYPE
   keyboard = [DEVICE] char;
```

Another situation when normal optimization techniques can change the meaning of a program is in loop–invariant expressions. For instance, using the keyboard example again, suppose you have the program segment:

```
TYPE
   keyboard = char;

VAR
   x  : integer;
   c  : char;
   kb : ^keyboard;
         .
         .
         .
   while (x < 10) do
      begin
      c := kb^;
      foo(c);
      x := x + 1;
      end;
```

The purpose of the block is to read 10 successive characters from the keyboard and pass each to a function called foo. However, to the compiler, it looks like an inefficient program since c will be assigned the same value 10 times. To optimize the program, the compiler may translate it as if it had been written:

```
   c := kb^;
   while (x<10) do
      begin
      foo(c);
      x := x + 1;
      end;
```

To ensure that the compiler does not optimize your program in that manner, declare kb as follows:

```
TYPE
   keyboard = [DEVICE] char;
VAR
   kb : ^keyboard;
```

In addition to suppressing optimizations, you can also use **device** to specify that a device is either exclusively read from or exclusively written to. You achieve this by using the read and write options which have the following meanings:

- **Device**(read) -- This attribute specifies read-only access for this variable or type. That is, if you attempt to write to this variable, the compiler flags the attempt as invalid and issues an error message. Although the syntax is available, the read and write options currently have no effect. They will be implemented in a future release of DOMAIN Pascal.
- **Device**(write) -- This attribute specifies write-only access for this variable or type. That is, if you attempt to read from this variable, the compiler flags the attempt as invalid and issues an error message. Although the syntax is available, the read and write options currently have no effect. They will be implemented in a future release of DOMAIN Pascal.
- **Device**(read, write) -- This attribute specifies both read and write access for this variable. This attribute is identical to the **device** attribute without any options.
- **Device**(write, read) -- Same as **device**(read, write).

For example, here are some sample declarations using the **device** attributes:

```
TYPE
   truth_array : [DEVICE] array[1..10] of boolean;
VAR
   c  : [DEVICE(read)] char;   {read-only access.}
   c2 : [DEVICE(write)] char;  {write-only access.}
   t  : truth_array;           {read and write access.}
```

3.14.3 Address -- Extension

Address takes one required argument.

The **address** specifier binds a variable to the specified virtual address, specified by a constant. You can only use **address** in a **var** declaration, not in a **type** declaration.

Address is useful for referencing objects at fixed locations in the address space (such as device registers, the PEB page, or certain system data structures). Typically, the compiler generates ABSOLUTE addressing modes when accessing such an operand. You cannot specify **define**, **extern**, or **static** when you use this option.

Using **address** by itself (without **device** or **volatile**) does not suppress any compiler optimizations. You should use it in conjunction with **volatile** or **device**. The example below associates the variable peb_page with the hexadecimal virtual address FF7000.

```
VAR
   peb_page : [ADDRESS(16#FF7000), DEVICE(read)]  char;
```

3.14.4 Attribute Inheritance -- Extension

Types and variables inherit the **device** attribute, and in some cases the **volatile** attribute, from more primitive data types. If you define a data type in terms of a more primitive data type declared with **device** or **volatile**, the new data type may inherit the attributes of that more primitive data type. For example, in the following declarations, `resource` inherits the **volatile** attribute from `semaphore`:

```
TYPE
   semaphore = [VOLATILE] integer;
   resource  = array[1..10] of semaphore;
```

If you define a record type as **volatile** or **device**, all the fields within the record inherit the attribute. And if you designate any one field within a record as having the **device** attribute, the entire record itself inherits the **device** attribute. However, the same is *not* true for a **volatile** field within a record; the entire record is not considered **volatile** just because one field is declared that way. Consider the following:

```
TYPE
   lock  = [VOLATILE] integer;
   queue = RECORD
           key : lock;
           users : integer;
               .
               .
               .
   end;
VAR
   wait : queue;
```

In this example, all references to `wait.key` are volatile, because the lock type is declared as **volatile**, but references to `wait.users` are not volatile. If you want all the fields to be volatile, insert the following after the record definition:

```
volque = [VOLATILE] queue;
```

> **NOTE:** Pointer types do not inherit the **device** or **volatile** attributes of their base type. However, when pointer variables are de-referenced, the system applies any attributes of their base type.

3.14.5 Special Considerations -- Extension

It is usually incorrect to associate an attribute with a pointer type. For example, declaring a pointer to a device register by means of the following declaration is almost certainly incorrect:

```
VAR
   iodata : [DEVICE] ^integer16;
```

The memory location of `iodata` is normally on the stack or in the DATA$ section. You don't want to make the local variable a device, you want to make the local variable a pointer to a device. Specify the following declarations instead:

```
TYPE
   DevInt = [DEVICE] integer;
VAR
   iodata : ^DevInt;
```

Chapter 4

Code

This chapter describes the statements, procedures, functions, and operators constituting the action part of a DOMAIN Pascal program or routine. The beginning of the chapter provides an overview of what's available. The remainder of the chapter is a DOMAIN Pascal encyclopedia complete with many, many examples. If you are a Pascal beginner, you should read a good Pascal tutorial textbook before trying to use this chapter.

The overview of DOMAIN Pascal is divided into the following categories:

- Conditional branching
- Looping
- Mathematical operators
- Input and output
- Miscellaneous functions and procedures
- Systems programming functions and procedures

4.1 Overview: Conditional Branching

DOMAIN Pascal supports the two standard Pascal conditional branching statements -- **if** and **case**.

4.2 Overview: Looping

DOMAIN Pascal supports **for**, **repeat**, and **while** -- the three looping statements of standard Pascal. All three looping statements support the **next** and **exit** extensions. **Next** causes a jump to the next iteration of the loop, and **exit** transfers control to the first statement following the end of the loop.

4.3 Overview: Mathematical Operators

DOMAIN Pascal supports all the standard arithmetic, logical, and set operators, as well as three additional operators for bit manipulation. Table 4–1 lists these operators.

Table 4–1. DOMAIN Pascal Operators

Type	Operator	Meaning
Integer	+	Addition
	–	Subtraction
	*	Multiplication
	/	Division (Real values)
	div	Division (Integer values)
	mod	Modulus arithmetic (returns remainder of an integer division)
Bit	&	Bitwise and
	!	Bitwise or
	~	Bitwise negation
Set	+	Set union
	*	Set intersection
	–	Set exclusion
	=	Set equality
	<>	Set inequality
	<=	First operand is subset of second
	>=	First operand is superset of second
	in	First operand is element of second
Boolean	**and**	Logical and
	or	Logical or
	not	Logical negation
All Types	=	Equal to
	<>	Not equal to
	>	Greater than
	>=	Greater than or equal to
	<	Less than
	<=	Less than or equal to

When evaluating expressions, DOMAIN Pascal uses the order of precedence rules found in Table 4–2. The expressions grouped together have the same precedence. Note that some operators work as both mathematical operators and as set operators. Nevertheless, the precedence rules are the same no matter how the operator is used.

Table 4–2. Order of Precedence in Evaluating Expressions

Operator	Order of Precedence
~ **not**	highest precedence
& * / **div** **mod** **and**	
! + – **or**	
= <> > >= < <= **in**	lowest precedence

DOMAIN Pascal permits the mixing of real and integer types in arithmetic expressions. For such mixed operations, DOMAIN Pascal promotes the integers to reals before performing the operation.

4.3.1 Expansion of Operands

The compiler computes operands smaller than 32 bits with 32 bits of precision when necessary to achieve correct arithmetic. This means **integer16** operands sometimes are expanded to **integer32** before calculations. These data expansions produce more accurate results; however, the compiler tries to avoid the extra code produced by data expansion where possible.

4.3.2 Predeclared Mathematical Functions

In addition to the mathematical operators, you can also use any of the predeclared mathematical functions listed in Table 4–3. Note that although the **arctan**, **cos**, **exp**, **ln**, **sin**, and **sqrt** functions permit integer arguments, the compiler converts an integer argument to a real number before calculating the function. Therefore, when possible, it is better to supply real, rather than integer, arguments to these functions.

Table 4–3. Mathematical Functions

Function	Argument(s)	Result	Meaning
abs(x)	int or real	same type as x	Absolute value of x.
arctan(x)	int or real	real	Arctangent of x.
arshft(x,n)	both are ints	int	Shifts the bits in x to the right n places. Preserves the sign of x.
cos(x)	int or real	real	Cosine of x.
exp(x)	int or real	real	Exponential function e raised to the x power.
ln(x)	int or real	real	Natural log of x; x > 0
lshft(x,n)	both are ints	int	Shifts the bits in x to the left n places.
odd(x)	int	boolean	True if x is an odd value.
round(x)	real	int	Round x up or down to nearest integer.
rshft(x,n)	both are ints	int	Shifts the bits in x to the right n places.
sin(x)	int or real	real	Sine of x.
sqr(x)	int or real	same type as x	Square of x.
sqrt(x)	int or real	real	Square root of x.
trunc(x)	real	int	Truncates x to nearest integer.
xor(x,n)	both are ints	int	Bit exclusive or.

4.4 Overview: I/O

DOMAIN Pascal supports the I/O procedures described in Table 4-4. For details on these routines, consult the encyclopedia later in this chapter and see Chapter 8.

Table 4-4. Predeclared I/O Procedures

Name	Action
close	Closes a file.
eof	Tests whether the stream marker is pointing to the end of the file.
eoln	Tests whether the stream marker is pointing to the end of a line.
find	Sets the stream marker to the specified record.
get	Reads from a file.
open	Opens a file for future access.
page	Inserts a formfeed (page advance) into a file.
put	Writes to a file.
read	Reads information from the specified file (or from the keyboard) into the specified variables. After reading the information, **read** positions the stream marker so that it points to the character or component immediately after the last character or component it read.
readln	Similar to **read** except that after reading the information, **readln** positions the stream marker so that it points to the character or component immediately after the next end-of-line character.
replace	Substitutes a new record component for an existing record.
reset	Specifies that an open file be open for reading only.
rewrite	Specifies that an open file be open for writing only, or tells the system to open a temporary file.
write	Writes the specified information to the specified file (or to the screen).
writeln	Same as **write** except that **writeln** always appends a linefeed to its output.

4.5 Overview: Miscellaneous Routines and Statements

Several DOMAIN Pascal elements do not fit neatly into categories. We've listed these elements in Table 4-5.

Table 4-5. Miscellaneous Elements

Element	Action
addr	Returns the address of the specified variable.
char	Finds the character whose ASCII value equals the specified number.
discard	Explicitly discards a computed value.
dispose	Deallocates the storage space that a dynamic record was using.
exit	Transfers control to the first statement following a **for**, **while**, or **repeat** loop.
firstof	Returns the first possible value of a type or a variable.
goto	Unconditionally jumps to the first command following the specified label.
in_range	Tells you whether the specified value is within the defined range of an enumerated variable.
lastof	Returns the last possible value of a type or a variable.
max	Returns the larger of two expressions.
min	Returns the smaller of two expressions.
new	Allocates space for storing a dynamic record.
next	Transfers control to the test for the next iteration of a **for**, **while**, or **repeat** loop.
nil	A special pointer value that points to nothing.
ord	Finds the ordinal value of a specified integer, Boolean, enumerated, or char data type.
pack	Copies unpacked array elements to a packed array.
pred	Finds the predecessor of a specified value.
return	Causes program control to jump back to the calling procedure or function.
sizeof	Returns the size (in bytes) of the specified data type.
succ	Finds the successor of a specified value.
type transfer functions	Permits you to change the data type of a variable or expression in the code portion of your program.
unpack	Copies packed array elements to an unpacked array.
with	Lets you abbreviate the name of a record. **With** is standard, but DOMAIN Pascal includes an extension that supports a name tag.

4.6 Overview: Systems Programming Routines

Several DOMAIN Pascal routines are available for systems programmers' use. Table 4–6 lists these routines. Because only a few programmers will need to use these routines, they are not described in the encyclopedia section that follows. Instead, they appear in Appendix E.

Table 4–6. Systems Programming Routines

Routine	Action
disable	Turns off the interrupt enable in the hardware status register.
enable	Turns on the interrupt enable in the hardware status register.
set_sr	Saves the current value of the hardware status register and then inserts a new one.

4.7 Encyclopedia of DOMAIN Pascal Code

The remainder of this chapter contains an alphabetical listing of each of the keywords that you can use in the action part of a DOMAIN Pascal program or routine. It also contains listings for several other Pascal concepts. Table 4–7 provides the keyword listings, and Table 4–8 contains the conceptual listings.

Table 4–7. Keyword Listings in Encyclopedia

abs	**get**	**pred**
addr	**goto**	**put**
and	**if then else**	**read**
arctan	**in**	**repeat**
arshft	**in_range**	**replace**
begin	**lastof**	**reset**
case	**ln**	**return**
chr	**lshft**	**rewrite**
close	**max**	**round**
cos	**min**	**rshft**
discard	**mod**	**sin**
dispose	**new**	**sizeof**
div	**next**	**sqr**
end	**nil**	**sqrt**
eof	**not**	**succ**
eoln	**odd**	**trunc**
exit	**open**	**unpack**
exp	**or**	**while**
find	**ord**	**with**
firstof	**pack**	**write**
for	**page**	**xor**

Table 4–8. Conceptual Listings in Encyclopedia

array operations	record operations
bit operations	set operations
compiler directives	statements
expressions	type transfer functions
pointer operations	

Abs -- Returns the absolute value of an argument.

FORMAT

abs(number) {**abs** is a function.}

Arguments

number	Any real or integer expression.

Function Returns

The **abs** function returns a real value if number is real and an integer value if number is an integer.

DESCRIPTION

The **abs** function returns the absolute value of the argument. The absolute value is the number's distance from 0. Note that number cannot be -2147483648 (which is the lowest negative integer).

EXAMPLE

```
program abs_example;
VAR
   x  : INTEGER;
   y  : REAL;

BEGIN
   x := -3;        x := ABS(x);
   y := -456.78;   y := ABS(y);
   WRITELN(x,y);
END.
```

Using This Example

If you execute the sample program named abs_example, you get the following output:

```
3     4.567800E+02
```

Addr -- Returns the address of the specified variable. (Extension)

FORMAT

addr(x) {**addr** is a function.}

Argument

x	Can be a variable declared as any data type except as a procedure or function data type having the **internal** attribute. X can also be a string constant but it cannot be a constant of any type other than string.

Function Returns

The **addr** function returns an **univ_ptr** value. (Chapter 3 describes the **univ_ptr** data type.)

DESCRIPTION

Use **addr** to return the address at which variable x is stored. **Addr** is of particular use with variables defined as pointers to functions or procedures.

Using **addr** can prevent some compiler optimizations. If you apply **addr** to a variable that is local to a routine, and the variable is not a **set, record,** or **array,** you do not get optimizations and register allocation for that variable or any expressions using the variable. This means the routine's code might be larger and slower than it otherwise would be.

Applying **addr** to a variable is equivalent to declaring the variable **volatile.** See Chapter 3 for more information on **volatile.**

Refer to the "Pointer Operations" listing later in this chapter for an example of **addr.**

EXAMPLE

```
Program addr_example;

TYPE
    ptr_to_real = ^real;

VAR
    y, y2        : real;
    ptr_to_y     : ptr_to_real;

BEGIN
    write ('Enter a real number -- '); readln(y);
{ Set ptr_to_y to the address at which y is stored. }
    ptr_to_y := ADDR(y);
{ Set y2 to the contents stored at y's address; i.e., set y2 equal to y. }
    y2 := ptr_to_y^;
    writeln(y2);
END.
```

Using This Example

Following is a sample run of the program named addr_example:

```
Enter a real number -- 5.3
    5.300000E+00
```

And -- Calculates the logical and of two Boolean arguments.

FORMAT

x **and** y {**and** is an operator.}

Arguments

x, y	Any Boolean expressions.

Operator Returns

The result of an **and** operation is a Boolean value.

DESCRIPTION

Sometimes **and** is called Boolean multiplication. Use it to find the logical and of expressions x and y. Here is the truth table for **and**:

x	y	**Result**
true	true	true
true	false	false
false	true	false
false	false	false

(See also the listings for the logical operators **or** and **not** later in this encyclopedia).

NOTE: Some programmers confuse **and** with "&". "&" is a bit operator; it causes DOMAIN Pascal to perform a logical and on all the bits in its two arguments. For example, compare the following results:

the result of (true **and** false) is false
the result of (75 & 15) is 11

(Refer to "Bit Operators" later in this encyclopedia.)

NOTE: Don't confuse **and** with the **and then** option of the **if** statement. Refer to the **if** listing later in this encyclopedia.

EXAMPLE

```
Program and_example;

CONST
    g = 6.6732e-11;

VAR
    mass1, mass2, radius, force : single;

BEGIN
   writeln('This program finds the gravitational force between two objects.');
   write('Enter the mass of the first object (in Kg) -- '); readln(mass1);
   write('Enter the mass of the second object (in Kg) -- '); readln(mass2);
   write('Enter the dist. between their centers (in M) -- '); readln(radius);

   if (mass1 > 0.0) AND (mass2 > 0.0)
        then force := (g * mass1 * mass2) / sqr(radius)
        else begin
               writeln('The data you have entered seems inappropriate');
               return;
             end;
   writeln('The force between these two objects is ', force:9:7, ' N');
END.
```

Using This Example

This program is available on-line and is named and_example.

Arctan -- Returns the arctangent of a specified number.

FORMAT

arctan(number) {**arctan** is a function.}

Argument

number	Any real or integer expression.

Function Returns

The **arctan** function returns a real value for the angle in radians.

DESCRIPTION

The **arctan** function returns the arctangent (in radians) of number. The arctangent of a number has the following relationship to the tangent:

```
y = arctan(x) means that x = tan(y)
```

Note that Pascal does not support a predeclared tangent function. However, you can find tangent(x) by dividing **sin**(x) by **cos**(x).

EXAMPLE

```
PROGRAM arctan_example;

{ This program demonstrates the ARCTAN function. }

CONST
   degrees_per_radian = 180.0 / 3.14159;

VAR
   q, answer_in_radians : REAL;
   answer_in_degrees    : INTEGER16;

BEGIN
   q := 2.0;

{First, find the arctangent of 2.0 in radians. }
   answer_in_radians := ARCTAN(q);
   writeln('The arctan of ', q:5:3, ' is', answer_in_radians:6:3, ' radians');

{Now, convert the answer to degrees. }
   answer_in_degrees := round(answer_in_radians * degrees_per_radian);
   writeln('The arctan of ', q:5:3, ' is ', answer_in_degrees:1, ' degrees');
END.
```

Using This Example

If you execute the sample program named arctan_example, you get the following output:

```
The arctan of 2.000000E+00 in radians is 1.107149E+00
The arctan of 2.000000E+00 in degrees is 63
```

Array Operations

Chapter 3 explains how to declare and initialize an array. In this listing, we explain how to use arrays in the code portion of your program.

ASSIGNING VALUES TO ARRAYS

To assign a value to an array variable, you must supply the following information:

- The name of the array variable.
- An index expression enclosed in brackets. The value of the index expression must be within the declared subrange of the index type.
- A value of the component type.

For example, the following program fragment assigns values to three arrays:

```
TYPE
  {elements is an enumerated type.}
   elements = (H, He, Li, Be, B, C, N, O, Fl, Ne);
   student  = record
       id   : integer16;
       class : (freshman, sophomore, junior, senior);
   end;

VAR
  {Here are three array declarations.}
   test_data      : array[1..100] of INTEGER16;
   atomic_weights : array[H..Be] of REAL;
   lie_test       : array[1..4, 1..2] of BOOLEAN; {2-dimensional array}
   class          : array[1..500] of student;

BEGIN
   test_data[37]       := 9018;
   atomic_weights[He]  := 4.0;
   lie_test[3, 2]      := true;
   student[30].id      := 8245;
   student[30].class   := senior;
      .
      .
      .
```

There are a few exceptions to the rule that you must supply an index expression.

The first exception is that you can assign a string to an array of **char** variable without specifying an index expression; for example, consider the following assignments to `greeting` and `farewell`:

```
CONST
   hi = 'aloha';

VAR
   greeting, farewell : array[1..12] of CHAR;

BEGIN
   greeting := hi;
   farewell := 'a bientot';
```

The only restriction on this kind of assignment is that the number of bytes in the string must be less than or equal to the declared number of declared components in the array. For example, you cannot assign the string 'auf wiedersehen' to `farewell` because the string contains 15 bytes and the array is declared as only 12 bytes. If you do try that assignment, the compiler reports:

```
Assignment statement expression is not compatible with the
assignment variable.
```

There is a second exception to the rule that you must specify an index expression when assigning a value to an array. The exception is that you can assign the value of one array to another array if both arrays are declared identically. For example, in the following program fragment, a and b are declared identically, though c and d are uniquely declared:

```
CONST
    quote = 'Ottawa!';

VAR
    a : array[1..20] of CHAR;
    b : array[1..20] of CHAR;
    c : array[1..21] of CHAR;
    d : array[1..19] of CHAR;

BEGIN
    a := quote;  {Assign the string 'Ottawa' to array a.     }
    b := a;  {This is a valid assignment.                    }
    c := a;  {This is not a valid assignment because a and c }
             { have different declared lengths.              }
    d := a;  {This is not a valid assignment because a and d }
             { have different declared lengths.              }
```

The assignment b := a causes DOMAIN Pascal to assign all components of array a to the corresponding indices in array b; that is, b := a is equivalent to the following 20 assignments:

```
b[1]  := a[1];
b[2]  := a[2];
  .        .
  .        .
  .        .
b[20] := a[20];
```

NOTE: In standard Pascal, before assigning a string to an array, you must explicitly pad the string to the length of the array. DOMAIN Pascal will pad the string for you, so you don't have to do it.

USING ARRAYS

You can specify an array component wherever you can specify a component variable of the same data type. In other words, if the compiler expects a real number, you can specify any real expression including a component of an array of real numbers.

NOTE: You can specify an array of **char** variable as an argument to **read, readln, write,** or **writeln.** However, you cannot specify any other component type of array as an argument to one of these procedures.

EXAMPLE

```
PROGRAM array_example;

{ This simple example reads in five input values, assigns the values to }
{ elements of an array, and then finds their mean.                      }

CONST
     number_of_elements = 5;

VAR
     a : array[1..number_of_elements] of single;
     running_total : single := 0.0;
     n : integer16;

BEGIN
    for n := 1 to number_of_elements do
       begin
          write('Enter a value -- ');
          readln(a[n]);
       end;

    for n := 1 to number_of_elements do
       running_total := running_total + a[n];

    writeln(chr(10), 'The mean is ', running_total/number_of_elements:3:1);

END.
```

Using This Example

Following is a sample run of the program named array_example:

```
Enter a value -- 4.3
Enter a value -- 10.3
Enter a value -- 9.5
Enter a value -- 6.2
Enter a value -- 1.5

The mean is 6.4
```

Arshft -- Shifts the bits in an integer to the right by a specified number of bits. Preserves the sign of the integer. (Extension)

FORMAT

arshft(num, sh) {**arshft** is a function.}

Arguments

num, sh	Must be integer expressions.

Function Returns

The function returns an integer value.

DESCRIPTION

Arshft does an arithmetic right shift of an integer. That is, it preserves the sign bit of num and then shifts the other bits sh positions to the right.

If num is a 16–bit integer and if the result of the function is to be stored in a 16–bit integer variable, then **arshft** expands num to a 32–bit integer, performs the shift, and then converts it back to a 16–bit integer.

First examine how **arshft** shifts a positive integer. Consider the effect of **arshft** on the 16–bit positive integer +100 in the following table:

```
unshifted              0000000001100100 = +100
ARSHFT(+100,1)         0000000000110010 = +50
ARSHFT(+100,2)         0000000000011001 = +25
ARSHFT(+100,3)         0000000000001100 = +12
```

Notice three things in the preceding table. First, the sign bit (the left–most bit) never changes. Second, notice that the bits move to the right. Third, notice that the bits do not wrap around from right to left; the absolute value always gets smaller.

Now, examine how **arshft** shifts a negative integer. Consider the effect of **arshft** on the 16–bit negative integer –100 in the following table:

```
unshifted              1111111110011100 = –100
ARSHFT(–100,1)         1111111111001110 = –50
ARSHFT(–100,2)         1111111111100111 = –25
ARSHFT(–100,3)         1111111111110011 = –13
```

In contrast to the preceding table, notice that **arshft** fills the left–most bits with ones rather than zeros as the right–most bits are shifted off the right end of the number.

Results are unpredictable if sh is negative.

EXAMPLE

```
PROGRAM arshft_example;

{ This program compares ARSHFT with RSFHT. }

VAR
    original_number, spaces_to_shift, r, ar : integer32 := 0;

BEGIN
   write('Enter a positive or negative integer -- '); readln(original_number);

   for spaces_to_shift := 1 to 5 do
      BEGIN
         writeln;
         writeln('When shifted ', spaces_to_shift:1, ' spaces.');

         r := RSHFT(original_number, spaces_to_shift);
         writeln('     The rshft result is ', r:1);

         ar := ARSHFT(original_number, spaces_to_shift);
         writeln('     The arshft result is ', ar:1);
      END;
END.
```

Using This Example

This program is available on-line and is named arshft_example.

Begin -- Marks the start of a compound statement.

FORMAT

Begin is a reserved word.

DESCRIPTION

Begin and **end** bracket a sequence of Pascal statements. A program must contain at least as many **ends** as **begins**. (Note that a program can contain more **ends** then **begins**.) You must use a **begin/end** pair to bracket a compound statement. (Refer to the "Statements" listing later in this encyclopedia.)

EXAMPLE

```
PROGRAM begin_end_example;

{This program does very little work, but does have lots of BEGINs and ENDs.}

TYPE
   student = record
      age : 6..12;
      id  : integer16;
   end;  {student record definition}

VAR
   x   : integer32;

PROCEDURE do_nothing;
BEGIN   {do_nothing}
   writeln('You have triggered a procedure that does absolutely nothing.');
   writeln('Though it does do nothing with elan.');
END;    {do_nothing}


FUNCTION do_next_to_nothing(var y : integer32) : integer32;
BEGIN  {do_next_to_nothing}
   do_next_to_nothing := abs(y);
END;   {do_next_to_nothing}


BEGIN  {main procedure}
   write('Enter an integer -- ');    readln(x);
   if x < 0
      then BEGIN
              writeln('You have entered a negative number!!!');
              writeln('Its absolute value is ', do_next_to_nothing(x):1);
           END
   else if x = 0
      then BEGIN
              writeln('You have entered zero');
              do_nothing;
           END
   else
              writeln('You have entered a positive number!!!');
END.   {main procedure}
```

Using This Example

This program is available on-line and is named begin_end_example.

Bit Operators -- Calculates "and", "or", and "not" on a bit by bit basis. (Extension)

FORMAT

```
op1 & op2            {& (an ampersand) is bit and.}
op1 ! op2            {! (an exclamation point) is bit or.}
~op1                 {~ (a tilde) is bit not.}
```

Arguments

op1, op2	Must be integer expressions.

Operator Returns

All three operators return integer results.

DESCRIPTION

DOMAIN Pascal supports three bit operators, all of which are extensions to standard Pascal. The operators perform operations on a bit by bit level using the following truth tables:

& (and)

bit x of op1	bit x of op2	bit x of result
0	0	0
0	1	0
1	0	0
1	1	1

! (or)

bit x of op1	bit x of op2	bit x of result
0	0	0
0	1	1
1	0	1
1	1	1

~ (not)

bit x of op1	bit x of result
0	1
1	0

Don't confuse these bit operators with the logical operators. Bit operators take integer operands; logical operators take Boolean operands.

In addition to the three bit operators, DOMAIN Pascal supports the following bit functions: **lshft**, **rshft**, **arshft**, and **xor**. All of these functions have their own listings in the encyclopedia.

NOTE: If one of the operators is declared as **integer32**, and the other operator is declared as **integer16**, DOMAIN Pascal extends the **integer16** to an **integer32** before calculating the answer.

NOTE: When performing these bitwise operations, DOMAIN Pascal treats the sign bit just as it treats any other bit.

EXAMPLE

```
PROGRAM bit_operators_example;

{ This program demonstrates bitwise AND, OR, and NOT. }

CONST
{ The 2# prefix specifies a base 2 number. }
    x = 2#0000000000001010; {10}
    y = 2#0000000000010111; {23}

VAR
    result1, result2, result3 : integer16;

BEGIN
    result1 := x & y;  writeln(x:1, ' AND ', y:1, ' = ', result1:1);
    result2 := x ! y;  writeln(x:1, ' OR ', y:1, ' = ', result2:1);
    result3 := ~x;     writeln('NOT ', x:1, ' = ', result3:1);
END.
```

Using This Example

If you execute the sample program named `bit_operators_example`, you get the following output:

```
10 AND 23 = 2
10 OR 23 = 31
NOT 10 = -11
```

Case -- A conditional branching statement that selects among several statements based on the value of an ordinal expression.

FORMAT

There are two different forms of the **case** statement. Here, we describe the use of **case** in the body of your program. The other use of **case** is in the variable or type declaration portion of the program. (See the "Variant Records" section in Chapter 3 for details on this use.)

Case takes the following syntax:

```
case expr of                              {case is a statement.}
        constantlist1 : stmnt1;
             .          .
             .          .
             .          .
        constantlistN : stmntN;
        otherwise  stmnt_list;
end;
```

Arguments

expr	Any ordinal expression (variable, constant, etc.) The ordinal types are integer, Boolean, char, enumerated, and subrange. You cannot specify an array as an expr, though you can specify an element of an array (assuming the element has an ordinal type). Also, you cannot specify a record, though you can specify a field of the record (assuming the field has an ordinal type).
constantlist	One or more values (separated by commas) having the same data type as expr.
stmnt	A simple statement or a compound statement (refer to the "Statements" listing later in this encyclopedia).
stmnt_list	One or more statements associated with the optional **otherwise** clause. (The **otherwise** clause tells the system to execute stmnt_list if expr matches none of the constants in any of the constantlists.) Unlike a compound statement, you do not have to bracket the stmnt_list with a **begin/end** pair (though doing so does not cause an error).

DESCRIPTION

The **case** statement performs conditional branching. It is very useful in situations involving a multi-way branch. When the value of expr equals one of the constants in a constantlist, the system executes the associated stmnt.

Note that **case** and **if/then/else** serve nearly identical purposes. The differences between **case** and **if/then/else** are:

- **Case** can only compare ordinal values. **If/then/else** can compare values of any data type.

- The system can sometimes execute a **case** statement faster than an equivalent **if/then/else** statement. That's because the DOMAIN Pascal compiler sometimes translates a **case** statement into a dispatch table and always translates an **if/then/else** statement into a series of conditional tests.

Also, note that a **case** statement is often more readable than an **if/then/else** statement. For instance, compare the following **if/then/else** statement to its equivalent **case** statement:

```
IF grade = 'A' THEN
   write('Excellent')
ELSE IF grade = 'B' THEN
   write('Good')
ELSE IF grade = 'C' THEN
   write('Average')
ELSE IF grade = 'D' THEN
   write('Poor')
ELSE IF grade = 'F' THEN
   write('Failing');
```

```
CASE grade OF
    'A' : write('Excellent');
    'B' : write('Good');
    'C' : write('Average');
    'D' : write('Poor');
    'F' : write('Failing');
end;
```

Otherwise -- Extension

DOMAIN Pascal supports an extension to the standard Pascal **case** statement. This extension is the **otherwise** clause. The **otherwise** clause tells the system to execute stmnt_list if expr matches none of the constants in any of the constantlists. For example, you can write the preceding **case** example like this:

```
CASE grade OF
   'A' : write('Excellent');
   'B' : write('Good');
   'C' : write('Average');
   'D' : write('Poor');
   OTHERWISE write('Failing');
end;
```

Notice that you do not put a colon (:) after the keyword **otherwise**.

As mentioned earlier, the **begin/end** pair is optional in an **otherwise** clause. Therefore, the following two **case** statements are equivalent:

```
CASE number OF
   1, 2, 3 : writeln('Good');
   OTHERWISE writeln('Great.');
             writeln('Encore.');
end;
```

```
CASE number OF
   1, 2, 3 : writeln('Good');
   OTHERWISE begin
                writeln('Great');
                writeln('Encore');
             end;
end;
```

EXAMPLE

```
PROGRAM case_example;
VAR
    a_letter : char;
    sale     : boolean;
    price    : array[1..5] of char;

BEGIN
    write('Is whole wheat bread on sale today? -- ');
    readln(a_letter);

    CASE a_letter OF
        'y', 'Y'  : sale := true;
        'n', 'N'  : begin
                        sale := false;
                        writeln('Remember to tell them it''s organic.');
                    end;
        OTHERWISE   begin
                        writeln('You have made a mistake');
                        writeln('Please rerun the program');
                        return;
                    end;
    end; {CASE}

    if sale then
        price := '$1.99'
    else
        price := '$2.99';
    writeln('Mark it as ', price:5);

END.
```

Using This Example

This program is available on-line and is named case_example.

Chr -- Returns the character whose ASCII value corresponds to a specified ordinal number.

FORMAT

chr(number) {**chr** is a function.}

Argument

number	An integer.

Function Returns

The **chr** function returns a value with the **char** data type.

DESCRIPTION

The **chr** function returns the character that has an ASCII value equal to the value of the low eight bits of number. Appendix B contains an ASCII table.

Chr produces a character with the bit pattern

```
number & 16#FF
```

Usually, number is between 0 and 127, in which case the character that **chr** returns is simply the character that has the ASCII value of number. If number is greater than 127, **chr** returns the character having the ASCII value of

```
number MOD 256
```

See the **mod** listing later in this encyclopedia.

Note that the **ord** function is the inverse of **chr** when **ord**'s argument type is **char**. (See the **ord** listing later in this encyclopedia.)

EXAMPLE

```
PROGRAM chr_example;

{ This program demonstrates three uses for the CHR function.               }

VAR
    capital_letter : 65..90;
    y : CHAR;
    age : 10..99;
    c_array : array[1..2] of char;

BEGIN
{ First, we'll use CHR to convert an integer to its ASCII value. }
    write('Enter an integer from 65 to 90 -- ');
    readln(capital_letter);
    y := CHR(capital_letter);
    writeln(capital_letter:1, ' corresponds to the ', y:1, ' character');
    writeln;

{ Second, we'll use CHR to ring the node bell. }
    write(chr(7));

{ The graphics primitive function gpr_$text writes character arrays to the  }
{ display.  But suppose you want gpr_$text to write an integer. In order to }
{ accomplish this task, you would write a routine similar to the following  }
{ which converts a 2-digit integer into a 2-character array.  Note that     }
{ 48 is the ASCII value for the '0' character, 49 for the '1' character,    }
{ and so on up to 57 for the '9' character.                                 }
    write('Enter an integer from 10 to 99 -- '); readln(age);
    c_array[1] := CHR((age DIV 10) + 48);
    c_array[2] := CHR((age MOD 10) + 48);
    writeln('The first digit is ', c_array[1]:1);
    writeln('The second digit is ', c_array[2]:1);
    writeln('The entire array is ', c_array);
END.
```

Using This Example

Following is a sample run of the program named chr_example:

```
Enter an integer from 65 to 90 -- 83
83 corresponds to the S character

Enter an integer from 10 to 99 -- 71
The first digit is 7
The second digit is 1
The entire array is 71
```

Close -- Closes the specified file. (Extension)

FORMAT

close(filename) {**close** is a procedure.}

Argument

filename	A file variable.

DESCRIPTION

Use the **close** procedure to close the file filename that you opened with the **open** procedure. By closing, we mean that the operating system unlocks it. When a program terminates (naturally or as a result of a fatal error), the operating system automatically closes all open files. So the **close** procedure is optional.

You cannot close the predeclared files **input** and **output**, but if you try, DOMAIN Pascal does not issue an error.

If filename is a temporary file, `CLOSE(filename)` deletes it.

Please see Chapter 8 for an overview of I/O.

NOTE: For permanent text files, your program should issue a **writeln** to the file just before closing it in order to flush the file's internal output buffer. If you don't include that **writeln**, the last line of the file may not be written.

EXAMPLE

```
PROGRAM close_example;
{ This program demonstrates the CLOSE procedure. }

CONST
    pathname = 'primates';

VAR
    class  : text;    {a file variable}
    name   : array[1..20] of char;
    status : integer32;

begin
    writeln('This program writes data to file "primates"');

    open(class, pathname, 'NEW', status);       {Open a file for writing.}
    if status = 0 then
        rewrite(class)
    else
        return;

    writeln('Enter the names of the children in your class -- ');
    writeln('The last entry should be "end"');
    repeat
        readln(name);
        if name <> 'end' then
            writeln(class, name)
        else
            exit;
    until false;

    CLOSE(class);           {Close the file for writing.}

{    . . .   }
{ Execute some time-consuming routines that do not access 'primates'. }
{    . . .  }



{Now, re-open the file for reading.}
    open(class, pathname, 'OLD', status);
    reset(class);

    writeln;
    writeln('Here are the names you entered:');
    repeat
        readln(class, name);
        writeln(name);
    until eof(class);

    CLOSE(class);
end.
```

Using This Example

This program is available on-line and is named close_example.

Compiler Directives -- Specify a variety of special services including conditional compilation and include files. (Extension)

FORMAT

The DOMAIN Pascal compiler understands the directives shown in Table 4-9. All directives begin with a percent sign (%). You can specify a directive anywhere a comment is valid. To use a directive, specify its name inside a comment or as a statement. For example, all of the following formats are valid:

```
{%directive}
(*%directive*)
%directive
```

If you specify a directive within a comment, the percent sign must be the first character after the delimiter (where spaces count as characters). In addition, you do not need to put a semicolon at the end of the directive.

You *must* place a semicolon after some directives if you use them as statements. Those directives are:

- **%debug;**
- **%eject;**
- **%include** 'pathname';
- **%list;**
- **%nolist;**
- **%slibrary** 'pathname';

Table 4-9. Compiler Directives

Directives	Action
☆ **%config**	Lets you easily set up a warning message if you forget to compile with the **-config** compiler option.
%debug;	Directs DOMAIN Pascal to compile lines prefixed by this directive when you use the **-cond** compiler option. If you don't use **-cond** when you compile, lines prefixed with **%debug** don't get compiled.
%eject;	Directs DOMAIN Pascal to put a formfeed in the listing file at this point.
☆ **%else**	Specifies that a block of code should be compiled if the preceding **%if** predicate **%then** is false.
☆ **%elseif** predicate **%then**	Directs the compiler to compile the code up until the next **%else**, **%elseif**, or **%endif** directive, if and only if the predicate is true.
☆ **%elseifdef** predicate **%then**	Checks whether additional predicates have been declared with a **%var** directive.
☆ **%enable;**	Sets compiler directive variables to true.
☆ **%endif**	Marks the end of a conditional compilation area of the program.
☆ **%error** 'string'	Prints 'string' as an error message whenever you compile.
☆ **%exit**	Directs the compiler to stop conditionally processing the file.
☆ **%if** predicate **%then**	Directs the compiler to compile the code up until the next **%else**, **%elseif**, or **%endif** directive, if and only if the predicate is true.
☆ **%ifdef** predicate **%then**	Checks whether a predicate was previously declared with a **%var** directive.
%include 'pathname';	Causes DOMAIN Pascal to read input from the specified file.
%list;	Enables the listing of source code in the listing file.

☆ is a directive described in "DIRECTIVES ASSOCIATED WITH THE -CONFIG OPTION."

Table 4-9. Compiler Directives (continued)

Directives	Action
%nolist;	Disables the listing of source code in the listing file.
%slibrary 'pathname';	Causes DOMAIN Pascal to incorporate a precompiled library into the program.
☆ **%var**	Lets you declare variables that you can then use as predicates in compiler directives.
%warning 'string'	Prints 'string' as a warning message whenever you compile.

☆ is a directive described in "DIRECTIVES ASSOCIATED WITH THE -CONFIG OPTION."

DIRECTIVES ASSOCIATED WITH THE -CONFIG OPTION

This subsection describes the following compiler directives: **%if, %then, %elseif, %else, %endif, %ifdef, %elseifdef, %var, %enable, %config, %error, %warning,** and **%exit.**

The conditional directives mark regions of source code for conditional compilation. This feature allows you to tailor a source module for a specific application. You invoke conditional processing by using the **-config** option when you compile. Unlike the other compiler directives, conditional directives cannot be used as comments.

Several of the directives take a predicate. A predicate can consist of special variables (declared with the **%var** directive) and the optional Boolean keywords **not, and,** or **or.** Given that `color` and mono are special variables, here are some possible predicates:

- `color`
- `NOT(color)`
- `mono OR color`
- `(mono AND color)`

%If predicate %then

If the predicate is true, DOMAIN Pascal compiles the code after **%then** and before the next **%else, %elseif,** or **%endif** directive.

For example, to specify that a block of code is to be compiled for a color node, you might choose an attribute name such as color to be the predicate. Then write:

```
%VAR color  {Tell the compiler that ´color´ can be used in a predicate.}
...

%IF color %THEN

   Code

%ENDIF;
```

To set color to true, you can either use the **%enable** directive in your source code or the **-config** option in your compile command line.

%Else

The **%else** directive is used in conjunction with **%if** predicate **%then. %Else** specifies a block of code to be compiled if the predicate in the **%if** predicate **%then** clause evaluates to false. For example, consider the following fragment:

```
%VAR color  {Tell the compiler that ´color´ can be used in a predicate.}
...

%IF color %THEN

   Code

%ELSE       {Compile this code if color is false.}

   Code

%ENDIF;
```

%Elseif predicate %then

%Elseif predicate **%then** is used in conjunction with **%if** predicate **%then**. It serves an analogous pupose to the Pascal statement

else if cond **then** statement

For example, suppose you want to compile one sequence of statements if the program is going to run on a color node, and another sequence of statements if the program is going to run on a monochromatic node. To accomplish that, you could organize your program in the following way:

```
%VAR color mono {Tell the compiler that ´color´ and ´mono´ can be }
                {used in a predicate.                            }
...

%IF color %THEN     {Compile the following code if color is true.}

   Code for color nodes

%ELSEIF mono %THEN  {Compile the following code if mono is true.}

   Code for monochromatic nodes

%ENDIF;
```

To set color or mono to true, you can either use the **%enable** directive in your source code or the **-config** option in your compile command line. If color and mono are both true, DOMAIN Pascal compiles the code for color nodes since it appears first. Note that you can put multiple **%elseif** directives in the same block.

%Endif

The **%endif** directive tells the compiler where to stop conditionally processing a particular area of code.

%Ifdef predicate %then

Use **%ifdef** predicate **%then** to check whether a variable was already declared with a **%var** directive. If you accidentally declare the same variable more than once, DOMAIN Pascal issues an error message. **%Ifdef** is a way of avoiding this error message. **%Ifdef** is especially helpful when you don't know if an include file declares a variable.

For example, consider the following use of **%ifdef**:

```
%INCLUDE 'bitmap_init.ins';  {Source code that may or may not have used  }
                             {%VAR to declare the variable 'color'.      }

%IFDEF not(color) %THEN      {If color has not been declared }
    %VAR color               {with %VAR, declare it now.     }
%ENDIF;
```

NOTE: The difference between **%if** and **%ifdef** is the following. Variables in an **%if** predicate are considered true if you set them to true with **%enable** or **-config**; however, variables in an **%ifdef** predicate are considered true if they have been declared with **%var**.

%Elseifdef predicate %then

%Elseifdef is to **%ifdef** as **%elseif** is to **%if**. Use **%elseifdef** predicate **%then** to check whether or not additional variables were declared with **%var**; for example:

```
%INCLUDE 'bitmap_init.ins';  {Source code that may or may not have   }
                             {used %VAR to declare the variables     }
                             {'color' or 'mono.'                     }

%IFDEF not(color) %THEN      {If color has not been declared with    }
    %VAR color               {%VAR, declare it now.                  }

%ELSEIFDEF not(mono) %THEN   {If mono has not been declared with     }
    %VAR mono                {%VAR, declare it now.                  }

%ENDIF;
```

%Var

The **%var** directive lets you declare variable and attribute names that will be used as predicates later in the program. You cannot use a name in a predicate unless you first declare it with the **%var** directive. The following example declares the names `code.old` and `code.new` as predicates:

```
%VAR  code.old  code.new
```

The compiler preprocessor issues an error if you attempt to declare with **%var** the same variable more than once. (Use **%ifdef** or **%elseifdef** to avoid this error.)

%Enable

Use the **%enable** directive to set a variable to true. (**%Enable** and the **-config** compiler switch perform the same function.) You create variables with the **%var** directive. If you do not specify a particular variable in an **%enable** directive or **-config** switch, DOMAIN Pascal assumes that it is false. For example, the following example declares three variables `code.sr9`, `code.sr8`, and `code.sr7`, and sets `code.sr9` and `code.sr7` to true:

```
%VAR     code.sr9  code.sr8  code.sr7
%ENABLE code.sr9  code.sr7
```

The compiler preprocessor issues an error message if you attempt to set (with **%enable** or **-config**) the same variable to true more than once.

%Config

The **%config** directive is a predeclared attribute name. You can only use **%config** in a predicate. The DOMAIN Pascal preprocessor sets **%config** to true if your compiler command line contains the **-config** option, and sets **%config** to false if your compiler command line does not contain the **-config** option. The purpose of the **%config** directive is to remind you to use the **-config** option when you compile; for example:

```
%IF color %THEN
   . . .
   {This is the code for color nodes.}
   . . .
%ELSEIF mono %THEN
   . . .
   {This is the code for monochromatic nodes.}
   . . .
%ELSEIF %config %THEN
   %warning('You did not set color or mono to true.');
%ENDIF
```

NOTE: Do not attempt to declare **%config** in a **%var** directive.

%Error 'string'

This directive causes the compiler to print 'string' as an error message. You must place this directive on a line all by itself. For example, suppose you want the compiler to print an error message whenever you compile with the **-config** mono option. In that case, set up your program like this:

```
%VAR color mono

%IF color %THEN
   ...
   {Code for color node.}
   ...
%ELSEIF mono %THEN
   %ERROR 'I have not finished the code for a monochromatic node.'
%ENDIF
```

If you do compile with the **-config** mono option, DOMAIN Pascal prints out the following error message:

```
 (0011)   %ERROR 'I have not finished the code for a monochromatic node.'
******** Line 11:  Conditional compilation user error.
1 error, no warnings, Pascal Rev n.nn
```

NOTE: Because of the error, DOMAIN Pascal does not create an executable object.

%Warning 'string'

This directive causes the compiler to print 'string' as a warning message. You must place this directive on a line all by itself. For example, suppose you want the compiler to print a warning message whenever you forget to compile with the **-config** color option. In that case, set up your program like this:

```
%VAR color mono

%IF color %THEN
   ...
   {Code for color node.}
   ...
%ELSE
   %WARNING 'You forgot to use the -CONFIG color option.'
%ENDIF
```

Then, if you don't compile with the **-config** color option, DOMAIN Pascal prints out the following error message:

```
 (0011)       %WARNING 'You forgot to use the -CONFIG color option.'
******** Line 11: Warning:  Conditional compilation user warning.
No errors, 1 warning, Pascal Rev n.nn
```

A warning does not prevent the compiler from creating an executable object.

%Exit

%Exit directs the compiler to stop conditionally processing the file. For example, if you put **%exit** in an include file, DOMAIN Pascal only reads in the code up until **%exit**. (It ignores the code that appears after **%exit**.)

%Exit has no effect if it's in a part of the program that does not get compiled.

DIRECTIVES NOT ASSOCIATED WITH THE -CONFIG OPTION

The remaining compiler directives are not specifically associated with the **-config** compiler option.

%Debug;

The **%debug** directive marks source code for conditional compilation. The "condition" is the compiler switch **-cond**. If you do compile with the **-cond** switch, DOMAIN Pascal compiles the lines that begin with **%debug**. If you do not compile with the **-cond** switch, DOMAIN Pascal does not compile the lines that begin with **%debug**. The reason this directive is called **%debug** is that it can help you debug your program. For instance, consider the following fragment:

```
          value := data + offset;
%DEBUG;   writeln('Current value is ', value:3);
```

The preceding fragment contains one **%debug** directive. If you compile with the **-cond** option, then the system executes the writeln statement at runtime. If you compile without the **-cond** option, the system does not execute the writeln statement at runtime. Therefore, you can compile with the **-cond** option until you are sure the program works the way you want it to work, and then compile without the **-cond** option to eliminate the (now) superfluous writeln message.

The **%debug** directive applies to one physical line only, not to one DOMAIN Pascal statement. Therefore, in the following example, **%debug** applies only to the **for** clause. If you compile with **-cond**, DOMAIN Pascal compiles both the **for** statement and the **writeln** procedure. If you compile without **-cond**, DOMAIN Pascal compiles only the **writeln** procedure (and thus there is no loop).

```
%DEBUG;    FOR j := 1 to max_size do
               WRITELN(barray[j]);
```

If you **%debug** within a line, text to the left of the directive is always compiled, and text to the right of the directive is conditionally compiled.

%Eject;

The **%eject** directive does not affect the **.bin** file; it only affects the listing file. (The **-l** compiler option causes the compiler to create a listing file.) The **%eject** directive specifies that you want a page eject (formfeed) in the listing file. The statement that follows the **%eject** directive appears at the top of a new page in the listing file.

%Include 'pathname';

Use the **%include** directive to read in a file ('pathname') containing DOMAIN Pascal source code. This file is called an include file. The compiler inserts the file where you placed the **%include** directive.

Many system programs use the **%include** directive to insert global type, procedure, and function declarations from common source files, called insert files. The DOMAIN system supplies insert files for your programs that call system routines. The insert files are stored in the */sys/ins* directory; see Chapter 6 for details.

DOMAIN Pascal permits the nesting of include files. That is, an include file can itself contain an **%include** directive.

The compiler option **-idir** enables you to select alternate pathnames for insert files at compiletime. See Chapter 6 for details.

NOTE: This directive has no effect if it's in a part of the program that does not get compiled.

%List; and %nolist;

The **%list** and **%nolist** directives do not affect the **.bin** file, they only affect the listing file. (The **-l** compiler option causes the compiler to create a listing file.) **%List** enables the listing of source code in the listing file, and **%nolist** disables the listing of source code in the listing file. For example, the following sequence disables the listing of the two insert files, and then re-enables the listing of future source code:

```
...
%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/ios.ins.pas';
%LIST;
...
```

%List is the default.

%Slibrary 'pathname';

The **%slibrary** directive is analogous to the **%include** directive. While **%include** tells the compiler to read in DOMAIN Pascal source code, **%slibrary** tells it to read in previously-compiled code.

The **%slibrary** directive tells the compiler to read in a precompiled library residing at 'pathname'. The compiler inserts the precompiled library where you place the **%slibrary** directive. The compiler acts as if the files that were used to produce the precompiled library were included at this point, except that any conditional compilation will have already occurred during precompilation.

Precompiled libraries can only contain declarations; they may *not* contain routine bodies and may *not* declare variables that would result in allocating storage in the default data section, DATA$. This means the declarations must either put variables into a named section, or must use the **extern** variable allocation clause. See Chapter 3 for more information about named sections, and Chapter 7 for details on **extern**.

Use the **-slib** compiler option (described in Chapter 6) to precompile a library and then insert **-slib**'s result in 'pathname'. For example, if you create a precompiled library called `mystuff.ins.plb`, this is how to include it in your program:

```
%SLIBRARY 'mystuff.ins.plb';
```

Precompiled library pathnames by default end in **.plb**.

Cos -- Calculates the cosine of the specified number.

FORMAT

cos(number) {**cos** is a function.}

Argument

number	Any real or integer value in radians (not degrees).

Function Returns

The **cos** function returns a real value (even if number is an integer).

DESCRIPTION

The **cos** function calculates the cosine of number.

EXAMPLE

```
PROGRAM cos_example;

{ This program demonstrates the COS function. }

CONST
    pi = 3.1415926535;

VAR
    degrees : INTEGER;
    q, c1, c2, radians : REAL;

BEGIN
    q := 0.5;
    c1 := COS(q);  { Find the cosine of one-half radians. }
    writeln('The cosine of ', q:5:3, ' radians is ', c1:5:3);

{ The following statements show how to convert from degrees to radians. }
{ More specifically, they find the cosine of 14 degrees.}
    degrees := 14;
    radians := ((degrees * PI) / 180.0);
    c2 := COS(radians);
    writeln('The cosine of ', degrees:1, ' degrees is ', c2:5:3);
END.
```

Using This Example

If you execute the sample program cos_example, you get the following output:

```
The cosine of 0.500 radians is 0.878
The cosine of 14 degrees is 0.970
```

Discard -- Explicitly discards the return value of an expression. (Extension)

FORMAT

discard(exp) {**discard** is a procedure.}

Argument

exp	Any expression, including a function call.

DESCRIPTION

In its effort to produce efficient code, the compiler sometimes issues warning messages concerning optimizations it performs. Those optimizations might not be right for your particular situation. For example, if you compute a value but never use it, the compiler may eliminate the computation, or the assignment of the value, and issue a warning message.

However, there are times when you call a function for its side effects rather than its return value. You don't need the value, but a Pascal function must *always* return a value to retain legal program syntax. You can't eliminate the function call without adversely affecting your program, but if you don't use the value, the compiler's optimizer automatically discards the return value and issues a warning message.

Since you know the return value is useless, in such a case you may want to eliminate this particular warning message. DOMAIN Pascal's **discard** procedure explicitly throws away the value of its exp and so gets rids of the warning. For example, to call a function that returns a value in `arg1` without checking that value, use **discard** as follows:

```
DISCARD(my_function(arg1));
```

EXAMPLE

```
PROGRAM discard_example;

VAR
    payment, monthly_sal : real;

{  The following function figures out whether a user can afford the    }
{  mortgage payments for a given house based on the rule that no more }
{  than 28% of one's gross monthly income should go to housing costs. }

FUNCTION enough(in         payment : real;
                in out monthly_sal : real) : boolean;
VAR
    amt_needed : real;

BEGIN
writeln;
amt_needed := monthly_sal * 0.28;

if amt_needed < payment then
    begin
    enough := false;
    monthly_sal := payment / (0.28);
    writeln ('Your monthly salary needs to be ', monthly_sal:6:2);
    end
else
    begin
    enough := true;
    writeln('Amazing! You can afford this house.');
    end
END;             {end function enough}

BEGIN            {main program}
    write ('How much is the monthly payment for this house? ');
    readln (payment);
    write ('What is your gross monthly salary? ');
    readln (monthly_sal);

{ The function enough can change the value of the global variable }
{ monthly_sal, so the function call is important, but its return  }
{ value is not. DISCARD that return value.                         }

    DISCARD (enough(payment,monthly_sal));
END.
```

Using This Example

Following is a sample run of the program named discard_example:

```
How much is the monthly payment for this house? 928
What is your gross monthly salary? 2400

Your monthly salary needs to be 3314.29
```

Dispose -- Deallocates the storage space that a dynamic variable was using. (Also refer to the listing for New.)

FORMAT

Dispose is a predeclared procedure that takes one of two formats. The format you choose depends on the format you use to call the **new** procedure. If you create a dynamic variable with the short form of **new**, then you must use the short form for **dispose**, which is:

dispose(p)   {**dispose** is a procedure.}

If you create a dynamic variant record with the long form of **new**, then you must use the long form of **dispose**, which is:

dispose(p, tag1..*tagN*);

Arguments

tag	One or more constants. The number of constants in a **dispose** call must match the number of constants in the **new** call.
p	A variable declared as a pointer. After you call `DISPOSE(p)`, DOMAIN Pascal sets p to **nil**.

DESCRIPTION

If p is a pointer, then `DISPOSE(p)` causes Pascal to deallocate space for the occurrence of the record that p points to. Deallocating means that Pascal permits the memory locations occupied by the dynamic record to be occupied by a new dynamic record. For example, consider the following declarations:

```
TYPE
   employeepointer = ^employee;
   employee = record
      first_name : array[1..10] of char;
      last_name  : array[1..14] of char;
      next_emp   : employeepointer;
   end;

VAR
   current_employee : employeepointer;
```

To store employee records dynamically, call `NEW(current_employee)` for every employee. If an employee leaves the company, and you want to delete his or her record, you can call `DISPOSE(current_employee)`. **Dispose** returns the storage occupied by that record for reuse by a subsequent **new** call.

If you create a dynamic record using a long-form **new** procedure, then you must call **dispose** with the same constants. For example, if you create a dynamic record by calling `NEW(widget, 378, true)`, then to deallocate the stored record, you must call `DISPOSE(widget, 378, true)`.

Note that the **dispose** procedure merely deallocates the record. If this disconnects a linked list, then it is up to you to reset the pointers. If some other variable points to this record and another program uses **dispose** to deallocate the record, then you get erroneous results. The moral: use **dispose** carefully.

> NOTE: If you call `DISPOSE(p)` when p is **nil**, DOMAIN Pascal reports an error. It is also an error to call **dispose** when p points to a block of storage space that you already deallocated with **dispose**. Finally, if you use a pointer copy that points to deallocated space, the results are unpredictable.

EXAMPLE

For a sample program that uses **dispose**, refer to the **new** listing later in this encyclopedia.

Div -- Calculates the quotient (excluding the remainder) of two integers.

FORMAT

d1 **div** d2 {**div** is an operator.}

Arguments

d1, d2	Any integer expression.

Operator Returns

The result of a **div** operation is always an integer.

DESCRIPTION

The expression (d1 DIV d2) produces the integer (nonfractional) result of dividing d1 by d2. **Div** uses the division rules of standard mathematics regarding negatives. For example, consider the following results:

```
 9 DIV 3 is equal to 3               -9 DIV 3 is equal to -3
10 DIV 3 is equal to 3              -10 DIV 3 is equal to -3
11 DIV 3 is equal to 3              -11 DIV 3 is equal to -3
12 DIV 3 is equal to 4              -12 DIV 3 is equal to -4
13 DIV 3 is equal to 4              -13 DIV 3 is equal to -4

 9 DIV (-3) is equal to -3           -9 DIV (-3) is equal to 3
10 DIV (-3) is equal to -3          -10 DIV (-3) is equal to 3
11 DIV (-3) is equal to -3          -11 DIV (-3) is equal to 3
12 DIV (-3) is equal to -4          -12 DIV (-3) is equal to 4
13 DIV (-3) is equal to -4          -13 DIV (-3) is equal to 4
```

To find the remainder of an integer division operation, use the **mod** operator. (See the **mod** listing later in this encyclopedia.)

See the NOTE in the "Expressions" listing later in this encyclopedia for information on using binary and unary operators together.

EXAMPLE

```
PROGRAM div_example;
{ This program converts a 3-digit integer to a 3-character array.         }
{ Note that the character 0 has an ASCII value of 48, the character 1     }
{ has an ASCII value of 49, and so on up until the character 9 which has  }
{ an ASCII value of 57.                                                   }

VAR
    x         : 100..999;
    digits    : array[1..3] of char;

BEGIN
   write('Enter a three-digit integer -- ');
   readln(x);

   digits[1] :=chr(48 + (x DIV 100));
   x := x MOD 100;
   digits[2] := chr(48 + (x DIV 10));
   digits[3] := chr(48 + (x MOD 10));

   writeln(digits);
END.
```

Using This Example

This program is available on-line and is named div_example.

Do -- Refer to the For or While listings later in this encyclopedia.

Downto -- Refer to If later in this encyclopedia.

Else -- Refer to If later in this encyclopedia.

End -- Signifies the end of a group of Pascal statements.

FORMAT

End is a reserved word.

DESCRIPTION

End is the terminator for a sequence of Pascal statements. A Pascal program must contain at least as many **ends** as **begins**.

Pascal requires a **begin/end** pair to bracket a compound statement. (Refer to the "Statements" listing later in this encyclopedia.)

Pascal requires **end** (without an accompanying **begin**) in the following situations:

- To terminate a **case** command.
- To terminate a record declaration.

Inexperienced Pascal programmers often wonder whether or not to put a semicolon (;) after an **end** in an **if/then/else** statement. Just remember the following rule: never put a semicolon after an **end** if it appears before the reserved word **else**.

EXAMPLE

```
PROGRAM begin_end_example;

{This program does very little work, but does have lots of BEGINs and ENDs.}

TYPE
   student = record
      age : 6..12;
      id  : integer16;
   end;  {student record definition}

VAR
   x    : integer32;

PROCEDURE do_nothing;
BEGIN    {do_nothing}
   writeln('You have triggered a procedure that does absolutely nothing.');
   writeln('Though it does do nothing with elan.');
END;     {do_nothing}


FUNCTION do_next_to_nothing(var y : integer32) : integer32;
BEGIN  {do_next_to_nothing}
   do_next_to_nothing := abs(y);
END;    {do_next_to_nothing}


BEGIN  {main procedure}
   write('Enter an integer -- ');    readln(x);
   if x < 0
      then BEGIN
              writeln('You have entered a negative number!!!');
              writeln('Its absolute value is ', do_next_to_nothing(x):1);
           END
   else if x = 0
      then BEGIN
              writeln('You have entered zero');
              do_nothing;
           END
   else
              writeln('You have entered a positive number!!!');
END.   {main procedure}
```

Using This Example

This program is available online and is named begin_end_example.

Eof -- Tests the current file position to see if it is at the end of the file.

FORMAT

eof(*filename*) {**eof** is a function.}

Argument

filename	A file variable symbolizing the pathname of an open file. *Filename* is an optional argument. If you do not specify *filename*, DOMAIN Pascal assumes that the file is standard input (**input**).

Function Returns

The **eof** function returns a Boolean value.

DESCRIPTION

The **eof** function returns true if the current file position is at the end of file *filename*; otherwise, it returns false. With one exception, *filename* must be open for either reading or writing when you call **eof**. The one exception occurs when *filename* is **input**; for a description of this exception, see the "Interactive I/O" section in Chapter 8.

EXAMPLE

```
PROGRAM eof_example;

CONST
    title_of_poem = 'annabel_lee';

VAR
    poetry  : text;
    stat    : integer32;
    a_line  : string;

BEGIN

{Open file anabel_lee for reading.}

open(poetry, title_of_poem, 'OLD', stat);
if stat = 0 then
    reset(poetry)
else
    return;

{Read each line from the file and write each line to the screen. }
{Halt execution when end of file is reached.                      }

while not EOF(poetry) do
    begin
    readln(poetry, a_line);
    writeln(output, a_line);
    end;

END.
```

Using This Example

This program is available on-line and is named eof_example.

Eoln -- Tests the current file position to see if it is pointing to the end of a line.

FORMAT

eoln(*f*)	{**eoln** is a function.}

Argument

f	A variable having the **text** data type. *F* is optional; if you do not specify it, **eoln** tests the standard input (**input**) file.

Function Returns

The function returns a Boolean value.

DESCRIPTION

The **eoln** function returns true when the stream marker points to an end-of-line character; otherwise, with two exceptions, **eoln** returns false. The two exceptions are:

- **Eoln** causes a runtime error if *f* was not opened for reading (with **reset**) or for writing (with **rewrite**). However, you do not need to open **input** or **output** for reading or for writing. (See the "Interactive I/O" section in Chapter 8 for details on **input** and **output**.)

- **Eoln** causes a runtime error if `EOF(f)` is true.

EXAMPLE

```
PROGRAM eoln_example;

CONST
    title_of_poem = 'annabel_lee';

VAR
    poetry  : text;
    stat    : integer32;
    a_char  : char;

BEGIN

{ Open file annabel_lee for reading. }
    open(poetry, title_of_poem, 'OLD', stat);
    if stat = 0
        then reset(poetry)
        else return;

{ Read in the first line of the poem one character at a time,  }
{ and write each character to the screen.                      }

    repeat
          read(poetry, a_char);
          writeln(output, a_char);
    until EOLN(poetry);

END.
```

Using This Example

This program is available on-line and is named eoln_example.

Exit -- Transfers control to the first statement following a For, While, or Repeat loop. (Extension)

FORMAT

Exit is a statement that neither takes arguments nor returns values.

DESCRIPTION

Use **exit** to terminate a loop prematurely; that is, to jump out of the loop you're in. In nested loops, **exit** applies to the innermost loop in which it appears. You can use **exit** within a **for**, **while**, or **repeat** loop only. If **exit** appears elsewhere in a program, DOMAIN Pascal issues an error.

It is preferable to use **exit** for jumping out of a loop prematurely rather than **goto**. That's because **goto** inhibits some compiler optimizations that **exit** does not.

EXAMPLE

```
PROGRAM exit_example;

VAR
    i, j        : integer16;
    data        : real;
    geiger      : array[1..5, 1..3] of real := [[* of 0.0],[* of 0.0],];

BEGIN
for i := 1 to 4 do
    begin
    writeln;
    for j := 1 to 3 do
        begin
        writeln(chr(10), 'Enter the data for coordinates', i:2, ',', j:1);
        write('(or enter -1 to jump down to the next row) -- ');
        readln(data);
        if data = -1 then
            EXIT
        else
            geiger[i,j] := data;
        end; {for j}
    end; {for i}
END.
```

Using This Example

Following is a sample run of the program named exit_example:

```
Enter the data for coordinates 1,1
(or enter -1 to jump down to the next row) -- 1.2

Enter the data for coordinates 1,2
(or enter -1 to jump down to the next row) -- -1


Enter the data for coordinates 2,1
(or enter -1 to jump down to the next row) -- 3.2

Enter the data for coordinates 2,2
(or enter -1 to jump down to the next row) -- 1.2

Enter the data for coordinates 2,3
(or enter -1 to jump down to the next row) -- 4.3


Enter the data for coordinates 3,1
(or enter -1 to jump down to the next row) -- -1


Enter the data for coordinates 4,1
(or enter -1 to jump down to the next row) -- 1.3

Enter the data for coordinates 4,2
(or enter -1 to jump down to the next row) -- 4.2

Enter the data for coordinates 4,3
(or enter -1 to jump down to the next row) -- -5
```

Exp -- Calculates the value of e, the base of natural logarithms, raised to the specified power. (See also Ln.)

FORMAT

exp(number)	{**exp** is a function.}

Argument

number	Any real or integer expression.

Function Returns

The **exp** function returns a real value.

DESCRIPTION

The **exp** function returns e raised to the power specified by number.

e to 16 significant digits is 2.718281828459045.

Pascal does not support an exponentiation function. However, you can use the **exp** and **ln** functions to simulate exponentiation. Just remember the following formula:

```
a b  = EXP(b * LN(a))
```

See the **ln** listing later in this encyclopedia.

EXAMPLE

```
PROGRAM exp_example;
{This example demonstrates the use of EXP in calculating the}
{exponential growth of bacteria.                            }

CONST
   c1  = 0.3466;

VAR
   starting_quantity    : INTEGER;
   ending_quantity, elapsed_time : REAL;

BEGIN
   write('How many bacteria are there at zero hour? -- ');
   readln(starting_quantity);
   write('How many hours pass? -- ');
   readln(elapsed_time);

   ending_quantity := starting_quantity * EXP(c1 * elapsed_time);

   writeln('There will be approximately ', ending_quantity:1,' bacteria.');
END.
```

Using This Example

Following is a sample run of the program named exp_example:

```
How many bacteria are there at zero hour? -- 10500
How many hours pass? -- 5.6
There will be approximately 7.313705E+04 bacteria.
```

Expressions

Throughout this encyclopedia, we refer to expressions. Here, we define expressions. An expression can be any of the following:

- A constant declared in a **const** declaration part
- A variable declared in a **var** declaration part
- A constant value
- A function call
- Any one of the above preceded by a unary operator appropriate to its data type
- Any two of the above separated by a binary operator appropriate to their data types

You can organize expressions into more complex expressions with parentheses. For example, the **odd** function requires an integer expression as an argument. The following program fragment demonstrates several possible arguments to **odd**:

```
CONST
    century := 100;
VAR
    x, y : integer;
  result : boolean;

BEGIN
    . . .
    result := ODD(century);               {a constant}
    result := ODD(x);                     {a variable}
    result := ODD(15);                    {a value}
    result := ODD(sqrt(25));              {a function}
    result := ODD(x + y);                 {an operation}
    result := ODD((x * 3) + sqr(y));      {several operations}
    . . .
```

NOTE: You cannot follow a binary operator with a unary operator of lower precedence. For example, consider the following proper and improper expressions:

```
9 DIV -3    {improper expression}
9 DIV (-3)  {proper expression}
5 * -100    {improper expression}
5 * (-100)  {proper expression}
```

Table 4-2 in the beginning of this chapter shows the order of precedence of operators.

Find -- Sets the file position to the specified record. (Extension)

FORMAT

find(file_variable, record_number, error_status); {**find** is a procedure.}

Arguments

file_variable	Must be a variable having the **file** data type. File_variable cannot be a variable having the **text** data type.
record_number	Must be an integer between 1 and n or between −1 and −n, where 1 denotes the first record of the file and n denotes the last record.
error_status	Must be declared as a variable with the **integer32** data type. DOMAIN Pascal returns a hexadecimal number in error_status which has the following meaning:

```
0 -- no error or warning occurred.

greater than 0 -- an error occurred.

less than 0 -- a warning occurred.
```

NOTE: Your program is responsible for handling the error. We detail error handling in Chapter 9.

DESCRIPTION

Before reading this, make sure you are familiar with the description of I/O in Chapter 8.

When you open a file for reading, the operating system sets the stream marker to the beginning of the file. You can call **read** to move this stream pointer sequentially, or you can call **find** to move it randomly.

Before you can call **find**, you must have first opened the file symbolized by file_variable for reading. (See Chapter 8 for a description of opening files for reading.) When you call **find**, DOMAIN Pascal sets the stream marker to point to the record specified by record_number.

If you specify a record_number between 1 and n, where n is the number of records in the file, **find** locates that number record. If record_number is between −1 and −n, **find** counts backward from the end of the file to locate the proper record. For example, if there are five records in the file and you specify −4 for record_number, DOMAIN Pascal counts back four from the end of the file and retrieves record number 2.

If you specify zero for record_number, DOMAIN Pascal returns an error code in error_status.

If you specify a record_number that is one greater than the number of records stored in the file, DOMAIN Pascal does not return an error code, but does not change the stream marker either.

After executing a **find**, DOMAIN Pascal sets the stream marker to point to the beginning of the next record. For example, if record_number is 2, then after executing a **find**, DOMAIN Pascal sets the stream marker to point to record 3.

Frequently, programmers use the **find** procedure with the **replace** procedure (which is described later in this encyclopedia).

NOTE: The term "record", means one occurrence in a DOMAIN record-structured (rec) file, which may or may not be a DOMAIN Pascal record type.

EXAMPLE

```
PROGRAM find_and_replace_example;

{ This program demonstrates the FIND and REPLACE procedures. }

%NOLIST;  { We need these include files for error checking.  }
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%LIST;

CONST
    pathname = 'his101';

TYPE
    student = RECORD
                  name    : array[1..12] of char;
                  age     : integer16;
              END;

VAR
    class               : FILE OF student;
    a_student           : student;
    st                  : status_$t;
    more_corrections    : char;
    particular_record   : integer16 := 0;
    n                   : integer16;

PROCEDURE print_records;
BEGIN
    n := 0;
    writeln(chr(10), 'Here are the records stored in the file:');
    reset(class);
    repeat
        n := n + 1;
        read(class, a_student);
        writeln('record ', n:2, '      ', a_student.name, a_student.age);
    until eof(class);
END;

PROCEDURE correct_errors;
BEGIN
    write('Enter the number of the record you wish to change -- ');
    readln(particular_record);

    if particular_record = n+1 then
        writeln ('There are only ', n:2, ' records in the file.')
    else
        BEGIN
        FIND(class, particular_record, st.all);
        if st.code = 0 then
            BEGIN
            write('What should this name be -- '); readln(a_student.name);
            write('What should this age be -- ');  readln(a_student.age);
            class^ := a_student;
            REPLACE(class);
            END
        else if st.code = stream_$end_of_file then
            BEGIN
            write ('You specified a number greater than the number of ');
            writeln ('records in the file.');
            END
        else
            error_$print(st);
        END;

END;


BEGIN  {main procedure}
    open(class, pathname, 'OLD', st.all);
    if st.code = 0 then
        BEGIN
        repeat
        print_records;
        write('Do you want to correct any records? (enter y or n) -- ');
        readln(more_corrections);
        if more_corrections = 'y' then
            correct_errors
        else
            exit;
        until false;
        END
    else if st.code = stream_$name_not_found then
        writeln('Did you remember to run put_example to create his101?')
    else
        error_$print(st);

END.
```

Using This Example

Following is a sample run of the program named find_and_replace_example:

```
Here are the records you have entered:
record  1     Kerry                 28
record  2     Barry                 26
record  3     Jan                   25
Do you want to correct any records? (enter y or n) -- y
Enter the number of the record you wish to change -- 2
What should this name be -- Sandy
What should this age be -- 27

Here are the records you have entered:
record  1     Kerry                 28
record  2     Sandy                 27
record  3     Jan                   25
Do you want to correct any records? (enter y or n) -- n
```

Firstof -- Returns the first possible value of a type or a variable. (Extension)

FORMAT

firstof(x) {**firstof** is a function.}

Argument

x	Is either a variable or the name of a data type. The data type can be a predeclared DOMAIN Pascal data type, or it can be a user–defined data type. X cannot be a record, file, or pointer type.

Function Returns

The **firstof** function returns a value having the same data type as x.

DESCRIPTION

The **firstof** function returns the first possible value of x according to the following rules:

Data Type of x	**Firstof returns**
integer or **integer16**	–32767
integer32	–2147483647
char	The character represented by `CHR(0)` called **nul.**
boolean	False.
enumerated	The first (leftmost) identifier in the data type declaration.
array	The lower bound of the subrange that defines the array's size.

The **firstof** function is particularly useful for finding the first element of an enumerated type (as in the example).

EXAMPLE

```
PROGRAM firstof_lastof_example;

TYPE
   astronomers = (aristotle, galileo, newton, tycho, kepler);

VAR
   stargazers   : astronomers;

BEGIN

   writeln('The following is a list of great astronomers:');
   for stargazers := firstof(astronomers) to lastof(astronomers) do
      writeln(stargazers);

END.
```

Using This Example

If you execute the sample program named `firstof_lastof_example`, you get the following output:

```
The following is a list of great astronomers:
      ARISTOTLE
        GALILEO
         NEWTON
          TYCHO
         KEPLER
```

For -- Repeatedly executes a statement a fixed number of times.

FORMAT

```
for index_variable := start_exp to | downto stop_exp do
       stmnt;                                              {for is a statement}
```

Arguments

index_variable	Any variable declared as an ordinal type. The ordinal types are enumerated, subrange, integer, Boolean, and char. Note that index_variable cannot be a real number. As an extension to standard Pascal, DOMAIN Pascal permits the index_variable to be declared in a scope other than the scope of the routine immediately containing the **for** loop.
start_exp	An expression matching the type of the index_variable.
stop_exp	An expression matching the type of the index_variable.
stmnt	A simple statement or compound statement. (Refer to the "Statements" listing later in this encyclopedia.)

DESCRIPTION

For, repeat, and **while** are the three looping statements of Pascal. With **for,** you explicitly define both a starting and an ending value to the index_variable.

When executing a **for** loop, Pascal initializes the index_variable to the value of the start_exp, and then either increments (**to**) or decrements (**downto**) the value of the index_variable by 1 until its value equals that of the stop_exp. When the index_variable equals the value of the stop_exp, Pascal executes the statements in the loop one final time before exiting the loop.

If index_variable is an integer or subrange variable, **for** increments or decrements its value by 1 for each cycle. If index_variable is a char variable, then **for** increments or decrements its ASCII value by 1 for each cycle. If index_variable is an enumerated variable, then incrementing means selecting the next element in sequence and decrementing means selecting the preceding element. If index_variable is a Boolean, then true has a value greater than false.

The keyword **to** causes incrementing; the keyword **downto** causes decrementing.

If you want to jump out of a **for** loop prematurely (i.e., before the value of the index_variable equals the value of the stop_exp), you have the following choices:

- Execute an **exit** statement to transfer control to the first statement following the **for** loop.
- Execute a **goto** statement to transfer control to outside of the loop.

In addition to these measures, you can also execute a **next** statement to skip the remainder of the statements in the loop and proceed to the next iteration. Here are some tips for using the **for** statement:

- Within the stmnt, you are not allowed to change the value of the index_variable.

- If you set up a meaningless relationship between the start_exp and the stop_exp (for example, FOR X := 8 TO 5 or FOR X := 10 DOWNTO 20), Pascal does not execute the loop even once.

EXAMPLE

```
PROGRAM for_example;

VAR
    time, year, zeta : integer16 := 0;
    hurricanes  : (king, donna, cleo, betsy, inez);
    scores : array[1..5, 1..3] of integer16;
    i, j : integer16;

BEGIN

{If you do not use a BEGIN/END pair, FOR assumes that the loop consists of }
{the first statement following it. }
    FOR time := 1 TO 3 DO
        writeln(time);

{To create a loop consisting of multiple statements, enclose the loop in }
{a BEGIN/END pair. }
    FOR time := 21 TO 30 DO
       begin
       year := year + time;
       writeln(year:5);  { Write a running total. }
       end;

{Here's an example of DOWNTO. }
    FOR time := year DOWNTO (year - 100) DO
       zeta := zeta + (time * 3);

{Here's an example of an enumerated index_variable. }
    FOR hurricanes := donna TO inez DO
       writeln(hurricanes);

{And finally, we use nested FOR loops to load a 2-dimensional array. }
    FOR i := 1 TO 5 DO
        begin      {for i}
        FOR j := 1 TO 3 DO
            begin  {for j}
            write('Enter the score for player ',i:1,' game ',j:1,' -- ');
            readln(scores[i,j]);
            end;   {for j}
        writeln;
        end;       {for i}
END.
```

Using This Example

This program is available on-line and is named for_example.

Get -- Advances the stream marker to the next component of a file.

FORMAT

get(f) {**get** is a procedure.}

Argument

f	A variable having the **file** or **text** data type.

DESCRIPTION

If f is a **file** variable, calling **get** causes the operating system to advance the stream marker so that it points to the next record in the file. If f is a **text** variable, calling **get** causes the operating system to advance the stream marker so that it points to the next character in the file.

After calling **get** to advance the stream marker, you can use another statement to read in the data that the stream marker points to and assign it to a variable from your program. Therefore, the sequence for reading in data looks like the following:

```
GET(f);          { Advance the stream marker. }
variable := f^;  { Set variable equal to whatever the stream marker }
                 { points to.                                        }
```

For example, the following program fragment demonstrates input via the **get** procedure:

```
VAR
    primes   : file of integer16;
    poem     : text;
    a_number : integer16;
    a_letter : char;

BEGIN
    . . .
    GET(primes);
    a_number := primes^;  { Set a_number equal to next record in primes.  }
    . . .

    GET(poem);
    a_letter := poem^;    { Set a_letter equal to next character in poem. }
    . . .
```

Note that the two statements

```
GET(poem);
a_letter := poem^;    { Set a_letter equal to next character in poem.     }
```

are identical to the single statement

```
READ(poem, a_letter);
```

Also notice that unlike **read**, **get** allows you to save the contents of f^.

You must open f for reading (with **reset**) before calling **get**. If EOF(f) is true, calling GET(f) causes a "read past end of file" error trap.

EXAMPLE

```
PROGRAM get_example;

{ This program demonstrates the GET procedure.          }
{ File 'his101' must exist before you run get_example. }
{ To create 'his101', you must run put_example.         }

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%LIST;

CONST
    file_to_read_from = 'his101';

TYPE
    student =
       record
            name   : array[1..12] of char;
            age    : integer16;
       end;

VAR
    class           : file of student;
    a_student       : student;
    st              : status_$t;

BEGIN
{Open a file for reading.}
    open(class, file_to_read_from, 'OLD', st.all );
    if st.code = 0
      then reset(class)
      else if st.code = stream_$name_not_found
         then begin
                writeln('Did you forget to run put_example?');
                return;
              end
         else error_$print(st);

{Now that the rec file is open, read all the records from it. }
    repeat
       a_student := class^;
       GET(class);
       write(chr(10), a_student.name);
       writeln(a_student.age:2);
    until eof(class);
END.
```

Using This Example

This program is available on-line and is named get_example.

Goto -- Unconditionally jump to a specified label in the program.

FORMAT

goto lbl;   {**goto** is a statement.}

Argument

lbl	Is an unsigned integer or identifier that you have previously declared as a label. (For information on declaring labels, see the "Label Declaration Part" section in Chapter 2.)

DESCRIPTION

A **goto** statement breaks the normal sequence of program execution and transfers control to the statement immediately following lbl.

A declared lbl usually is local to the block in which it is declared. That is, if you know you declared a label, but the compiler still reports the following error, you must move your label declaration to the correct procedure or function:

```
(Name_of_Label) has not been declared in routine (name_of_routine)
```

It is illegal to use **goto** to jump inside a structured statement (for example, a **for**, **while**, **case**, **with**, or **repeat**) from outside that statement. This means a fragment like this produces an error:

```
if error_flag = true then
    goto cleanup;
          .
          .
          .
for i := 1 to 10 do
    begin
          . . .
cleanup:                    {WRONG!}
          . . .
    end;                    {close for statement}
```

It is illegal to jump into an **if/then/else** statement if you compile with the **–iso** option. See Chapter 6 for more details.

Gotos are useful for handling exceptional conditions (such as an unexpected end of file). However, the compiler usually generates better code if you use **exit** rather than **goto** to jump out of a loop prematurely.

Nonlocal **gotos**, whose target lbl is in the main program or some other routine at a higher level, have a great effect on the generated code. They generally shut off most compiler optimizations on the code near the target lbl. In order to produce the most efficient code, you should try to use **goto** as infrequently as possible.

NOTE: You cannot jump into a structured statement from outside that statement. For example, the following contains two jumps that are always wrong and one jump that is incorrect if you compile with the **-iso** switch:

```
PROGRAM bad_gotos;
VAR
    z,x,value : integer16;
    a_char    : char;
LABEL
    900;

PROCEDURE foo;
LABEL
    100;

BEGIN
for x := 1 to 100 do
    begin
    writeln ('value ', x);
100: write ('Enter a value ');
    readln(value);
    z := z + value;
    end;
GOTO 100;                    {ILLEGAL: cannot jump to a label inside}
                             {the for loop.                         }
END;


BEGIN
write ('Do you want to use the program? ');
readln (a_char);

if a_char = 'y' then
    GOTO 100                  {ILLEGAL: cannot jump to a label in    }
                              {another routine.                      }
else if a_char = 'n' then
    GOTO 900                  {ILLEGAL IF COMPILED WITH -ISO SWITCH:}
                              {cannot jump to a label that's inside }
                              {another statement.                   }
else if a_char = 'o' then
    writeln ('o is not a legal response')
else
    900: writeln ('ok, we won''t use the program');
END.
```

Note that you can use **goto** to jump directly from a nested routine to an outer routine. For example, procedure xxx issues a valid **goto** in the following program:

```
Program non_local_goto;
Label
     900;


Procedure xxx;
BEGIN
     . . .
     GOTO 900;
     . . .
END;


BEGIN
     . . .
900: writeln('back in main program.');
     . . .
END.
```

EXAMPLE

```
PROGRAM goto_example;

TYPE
    possible_values = 10..25;

VAR
    x : possible_values;

LABEL
    100;

BEGIN
    writeln('You will now enter the experimental data.', chr(10));
100:
    write('Please enter the obtained value for x -- ');
    readln(x);
    if in_range(x) then
        writeln('This value seems possible.')
    else
        begin
        writeln('This value seems suspicious.');
        GOTO 100;
        end;
END.
```

Using This Example

Following is a sample run of the program named goto_example:

```
You will now enter the experimental data.

Please enter the obtained value for x -- 35
This value seems suspicious.
Please enter the obtained value for x -- 17
This value seems possible.
```

If -- Tests one or more conditions and executes one or more statements according to the outcome of the tests.

FORMAT

You can use **if**, **then**, and **else** in the following two ways:

```
if cond then stmnt;                      {first form}

if cond then stmnt1 else stmnt2;         {second form}
```

Arguments

cond	Any Boolean expression.
stmnt	A simple statement or a compound statement. (Refer to the "Statements" listing in this encyclopedia.) Note that stmnt can itself be another **if** statement.

DESCRIPTION

The **if** and **case** statements are the two conditional branching statements of Pascal.

In an **if/then** statement, if cond evaluates to true, Pascal executes stmnt. If cond is false, Pascal executes the first statement following stmnt.

In an **if/then/else** statement, if cond is true, Pascal executes stmnt1. However, if cond is false, Pascal executes stmnt2.

You often use an **if** statement to evaluate multiple conditions. To do so, just remember that a stmnt can itself be an **if** statement. For example, consider the following **if** statement which evaluates multiple conditions:

```
IF age < 3 THEN
    price = 0.0
ELSE IF (age >= 3) AND (age <= 6) THEN
    price = 1.00
ELSE IF (age > 6)  AND (age <=12) THEN
    price = 2.00
ELSE
    price = 4.00;
```

Inexperienced Pascal programmers (and even some of the experienced ones) often forget where to put semicolons in an **if/then/else** statement. Just remember that you never put a semicolon immediately prior to an **else**.

"And Then" and "Or Else" -- Extension

The **if** statement of DOMAIN Pascal supports an **and then** and **or else** extension to standard Pascal. You can use **and then** and **or else** wherever you'd use **and** and **or** in a cond except that **and then** and **or else** cannot be contained inside parentheses.

When you use **and** or **or**, there is no guarantee that Pascal will evaluate the Boolean expressions of a cond in the order that you write them. For example, in the following **if/then** command, Pascal may test ((y DIV x) > 0) before it tests (x <> 0):

```
IF (x <>  0) AND ((y DIV x) > 0)
    THEN ...
```

However, using **and then** or **or else** guarantees that DOMAIN Pascal evaluates the Boolean expressions of a cond in the order that you write them. **And then** and **or else** also guarantee "short-circuit" evaluation; that is, at runtime, the system only evaluates as many expressions as is necessary. For example, if you change **and** to **and then**, you guarantee that DOMAIN Pascal checks that x does not equal 0 before actually dividing by x:

```
IF (x <>  0) AND THEN ((y DIV x) > 0)
    THEN ...
```

And then forces DOMAIN Pascal to evaluate Boolean expressions in textual order *until* one of them is false. On a false, DOMAIN Pascal skips the remaining Boolean expressions. If all of them are true, the cond is true.

Or else forces DOMAIN Pascal to evaluate Boolean expressions in textual order *until* one of them is true. If one of the Boolean expressions is true, DOMAIN Pascal skips over the remaining tests.

Note that **and** and **or** force DOMAIN Pascal to test *all* Boolean expressions in the cond. Thus, **and then** and **or else** can be more efficient than **and** and **or** in some cases.

Standard Pascal does allow you to ensure the order of Boolean evaluation in a cond; however, the DOMAIN Pascal extension is much easier. For example, compare the standard Pascal code on the left with its functional equivalent on the right:

Standard Pascal	*DOMAIN Pascal*

`IF c1 THEN`
`    IF c2 THEN`
`       s1;`
	`IF c1 AND THEN c2 THEN`
	`   s1;`

NOTE: When using **and then** or **or else**, you cannot enclose the conds in parentheses. For example, the following **if** statement causes an error:

```
IF ((c1 = c2) AND THEN (c3 = c4))
   . . .
```

EXAMPLE

```
PROGRAM if_example;
{This program demonstrates IF/THEN and IF/THEN/ELSE.}
VAR
    y, age, of_age, root_ratings  : integer16;
    tree    : (ficus, palm, poinciana, frangipani, jacaranda);
    grade   : char;

BEGIN
write('Enter an integer -- '); readln(y);                    {USAGE 1}
IF y < 0  THEN
    writeln('Its absolute value equals ', (abs(y)):3);

write('Enter an age -- '); readln(age);                      {USAGE 2}
IF age > 18 THEN
    writeln('  An adult')
ELSE
    begin
    of_age := 18 - age;
    writeln('  A minor for another ', of_age:1,' years.');
    end;

write('Enter a grade -- '); readln(grade);                   {USAGE 3}
IF (grade = 'A') OR (grade = 'B') THEN
    writeln('  Good work')
ELSE IF (grade = 'C') OR (grade = 'D') THEN
    begin
    writeln('  Satisfactory work');
    writeln('  Though improvement is indicated.');
    end
ELSE IF (grade = 'F') THEN
    writeln('  Failing work');

write('Enter the name of a tropical tree -- '); readln(tree);   {USAGE 4}
IF (tree = poinciana) OR (tree = jacaranda) THEN
    begin
    writeln('  Blossoms in June and July.');
    root_ratings := 9;
    end
ELSE IF tree = palm THEN
    root_ratings := 8
ELSE
    root_ratings := 2;

{ Imagine a pointer variable which may be NIL or valid.  If it is NIL,  }
{  you do not want to dereference it, because the compiler generates an }
{  access violation.  It would be wrong to code it in the following way }
{  because there is no way to be sure which expression the compiler     }
{  will evaluate first:                                                  }
{              IF (ptr <> NIL) AND (ptr^ = 100) THEN ...                 }
{  However, coding it the following way ensures the correct order:       }
{              IF (ptr <> NIL) AND THEN (ptr^ = 100) THEN ...            }
END.
```

Using This Example

Following is a sample run of the program named if_example:

```
Enter an integer -- -10
Its absolute value equals  10
Enter an age -- 13
  A minor for another 5 years.
Enter a grade -- B
  Good work
Enter the name of a tropical tree -- poinciana
  Blossoms in June and July.
```

In -- Evaluates an expression to see if it is a member of a specified set.

FORMAT

exp **in** setexp {**in** is a set operator.}

Arguments

setexp	A set expression.
exp	An expression of the same data type as the elements constituting the base type of setexp.

Operator Returns

The result of an **in** operation is always Boolean.

DESCRIPTION

Use **in** to determine if exp is an element in set setexp. **In** returns either true or false.

EXAMPLE

```
PROGRAM in_example;
{ This program prompts the user for a word, then counts the number of }
{  ordinary vowels (a, e, i, o, and u) in the word. }

VAR
    word             : array [1..20] of char := [* of ' '];
    count_of_vowels  : integer16 := 0;
    x                : integer16;

BEGIN
   write('Enter a word -- ');
   readln(word);
   for x := 1 to 20 do
      if word[x] IN ['a', 'e', 'i', 'o', 'u']
        then count_of_vowels := count_of_vowels + 1;
   writeln('This word contains ', count_of_vowels:1, ' ordinary vowels.');
END.
```

Using This Example

Following is a sample run of the program named in_example:

```
Enter a word -- computers
This word contains 3 ordinary vowels.
```

In_range -- Determines whether or not a specified value is within the defined integer subrange. (Extension)

FORMAT

in_range(x) {**in_range** is a function.}

Argument

x	A variable having a scalar (i.e., integer, Boolean, char, enumerated, or subrange) data type. For most practical purposes, x must be an enumerated or a subrange variable.

Function Returns

The **in_range** function returns a Boolean value.

DESCRIPTION

The **in_range** function returns true if the value of x is within its defined range; otherwise, it returns false. The following program fragment demonstrates a possible use of **in_range**. We want to use **in_range** in this example, because it generates very efficient code:

```
TYPE
   small_int = -7..7;
VAR
   x : integer16;
BEGIN
   readln(x);
   if IN_RANGE(small_int(x))
      then ...
      else ...
```

EXAMPLE

```
PROGRAM in_range_example;

TYPE
   possible_temperature_range = 48..97;

VAR
   air_temp    : possible_temperature_range;
   stop        : boolean;


BEGIN
   repeat
        write('Enter the current air temp. (in deg. fahrenheit) -- ');
        readln(air_temp);
        if not IN_RANGE(air_temp) then
             begin
                writeln('This temperature is out of the historical range.');
                writeln;
                stop := false;
             end
        else begin
                writeln('This value is within the historical range.');
                stop := true;
             end;
   until stop;
END.
```

Using This Example

Following is a sample ex´cution of the program named in_range_example:

```
Enter the current air temp. (in deg. fahrenheit) -- 100
This temperature is out of the historical range.

Enter the current air temp. (in deg. fahrenheit) -- 47
This temperature is out of the historical range.

Enter the current air temp. (in deg. fahrenheit) -- 52
```

Lastof -- Returns the last possible value of a type or a variable. (Extension)

FORMAT

lastof(x) {**lastof** is a function.}

Argument

x	Either a variable or the name of a data type. The data type can be a predeclared DOMAIN Pascal data type, or it can be a user-defined data type. X cannot be a record, file, or pointer type.

Function Returns

The **lastof** function returns a value having the same data type as x.

DESCRIPTION

The **lastof** function returns the final possible value of x according to the following rules:

Data Type	Lastof Returns
integer or **integer16**	32767
integer32	2147483647
char	A symbol indicating an unprintable character; however, `ORD(LASTOF(char))` returns 255.
boolean	True.
enumerated	The last (right-most) identifier in the data type declaration.
array	The upper bound of the subrange that defines the array's size.

The **lastof** function is particularly useful for finding the last value in an enumerated type.

EXAMPLE

See the example in the **firstof** listing earlier in this encyclopedia.

Ln -- Calculates the natural logarithm of a specified number.

FORMAT

ln(number)	{**ln** is a function.}

Argument

number	Any real or integer expression that evaluates to a positive number.

Function Returns

The **ln** function always returns a real value (even if number is an integer).

DESCRIPTION

The **ln** function returns the natural logarithm of number. Refer to the **exp** listing earlier in this encyclopedia for a practical definition involving **ln**.

EXAMPLE

```
PROGRAM ln_example;
{ Each radioactive isotope has a unique K constant.}
{ This program uses LN and empirical data to calculate the k constant.}
VAR
    starting_quantity, ending_quantity : real;
    elapsed_time, k : real;

BEGIN
    write('Enter the quantity at time zero -- ');
    readln(starting_quantity);
    write('Enter the elapsed time (t) -- ');
    readln(elapsed_time);
    write('Enter the quantity at time (t) -- ');
    readln(ending_quantity);
    k := LN(starting_quantity/ending_quantity) * (1.0 / elapsed_time);
    writeln('The k constant for this radioactive element is ', k);
END.
```

Using This Example

Following is a sample run of the program named ln_example:

```
Enter the quantity at time zero -- 1230
Enter the elapsed time (t) -- 47
Enter the quantity at time t -- 753
The k constant for this radioactive element is 0.010
```

Lshft -- Shifts the bits in an integer a specified number of bit positions to the left. (Extension)

FORMAT

lshft(num, sh) {**lshft** is a function.}

Arguments

num, sh	Integer expressions.

Function Returns

The **lshft** function returns an integer value.

DESCRIPTION

The **lshft** function shifts the bits in num to the left sh places. **Lshft** does not wrap bits around from the left edge to the right; instead, **lshft** shifts zeros in on the right. For example, consider the following results:

```
VAR
   i, n : INTEGER16;

BEGIN
   i := 2000;     { 2#0000011111010000 = 10#+2000  }
   LSFHT(i, 1);   { 2#0000111110100000 = 10#+4000  }
   LSHFT(i, 3);   { 2#0011111010000000 = 10#+16000 }
   LSHFT(i, 7);   { 2#1110100000000000 = 10#-6144  }
```

Results are unpredictable if sh is negative.

Compare **lshft** to **rshft** and **arshft**.

EXAMPLE

```
PROGRAM lshft_example;

VAR
   unshifted_integer, shifted_integer, shift_left, x : integer16;
   choice     : 0 .. 15;
   drink_info : array[0..6] of integer16 := [* of 0];

BEGIN

{In the following subroutine, LSHFT acts as a multiplier according to the}
{equation LSHFT(num,sh) = num * (2 to the sh power).  Beware of overflow }
{when you use LSHFT for this purpose.                                    }

     unshifted_integer := 15; {15 is 0000000000001111 in binary}
     shift_left := 3;
     shifted_integer := LSHFT(unshifted_integer, shift_left);
     writeln(unshifted_integer:5, ' times 2 to the ', shift_left:1,
                ' power = ', shifted_integer:1);

{The result will be 120.}
{120 is 0000000001111000 in binary.}


{You can also use LSHFT to pack information more effectively.  For     }
{example, suppose you asked 24 people to name their favorite soft      }
{drink from a list of 16 possibilities.  Since we can represent 16     }
{possibilities in 4 bits, we can store 4 people's responses in one     }
{16-bit word in the following manner:                                  }
{                                                                      }
{Bit #  0          3   4          7   8          11  12          15  }
{       ------------------------------------------------------------- }
{       | Response 1  | Response 2  | Response 3  | Response 4  | }
{       ------------------------------------------------------------- }
{                                                                      }
{Therefore, we will only need six 16-bit words to store the data      }
{rather than 24 16-bit words.  The following code uses the LSHFT      }
{function to accomplish this data reduction.                           }
     writeln;
     for x := 0 to 23 DO
        BEGIN
           write('Enter the preference (0-15) of client ',
                  x:1, ' -- ');
           readln(choice);
           drink_info[x div 4] := drink_info[x div 4] !
                                   LSHFT(choice, (4 * (x mod 4)));
           writeln(DRINK_INFO[x DIV 4]);
        END;
{You can also achieve this sort of packing with packed records.        }

END.
```

Using This Example

This program is available on-line and is named lshft_example.

Max -- Returns the larger of two expressions. (Extension)

FORMAT

```
max(exp1,exp2)                         {max is a function.}
```

Arguments

exp1, exp2	Any valid expression.

DESCRIPTION

DOMAIN Pascal's **max** function returns the larger of the two input expressions. Exp1 and exp2 must be the same type or must be convertable to the same type by Pascal's default conversion rules (for example, integer converted to a real).

If exp1 and exp2 are unsigned scalars or pointers, DOMAIN Pascal performs an unsigned comparison. (The scalar data types are integers, Boolean, character, and enumerated.) If they are **real**, **single**, or **double**, a floating-point comparison is done, while if they are signed integers, DOMAIN Pascal performs a signed comparison.

See also **min**.

EXAMPLE

```
program max_example;

var
    i : integer16;
    x,y,newx,newy,high,newhigh:real;

begin

{ This program allows the user to calculate interest yields on          }
{ amounts deposited in two savings accounts that have different         }
{ interest rates. The first pays 6.55 Annual Percentage Rate (APR),     }
{ and the second pays 8.75% APR. After figuring how much would be       }
{ in each account after five years, the program then rolls the money   }
{ in the account with the highest balance over into a Certificate       }
{ of Deposit account which pays 11% APR, and then the interest it       }
{ would earn in five more years is calculated.                          }

write ('Enter the initial deposit for account 1: ');
readln (x);
write ('Enter the initial deposit for account 2: ');
readln (y);

newx := x;
newy := y;

for i := 1 to 5 do              { Figure interest for first   }
    begin                       { five years on both accounts.}
    x := newx;
    newx := x + (x * 0.0655);
    y := newy;
    newy := y + (y * 0.0875);
    end;

writeln;
writeln (' Account 1 balance after 5 years at 6.55%: ', newx:7:2);
writeln (' Account 2 balance after 5 years at 8.75%: ', newy:7:2);

high := MAX (newx,newy);        { Find the maximum interest yield.}

newhigh := high;
for i := 1 to 5 do
    begin
    high := newhigh;
    newhigh := high + (high * 0.11);
    end;

writeln;
writeln (' Highest account after 5 more years in CDs at 11%: ', newhigh:7:2);

end.
```

Using This Example

Following is a sample run of the program named max_example:

```
Enter the initial deposit for account 1: 1000
Enter the initial deposit for account 2: 900

Account 1 balance after 5 years at 6.55%: 1373.31
Account 2 balance after 5 years at 8.75%: 1368.95

Highest account after 5 more years in CDs at 11%: 2314.10
```

Min -- Returns the smaller of two expressions. (Extension)

FORMAT

min(exp1,exp2) {**min** is a function.}

Arguments

exp1, exp2	Any valid expression.

DESCRIPTION

DOMAIN Pascal's **min** function returns the smaller of the two input operands. Exp1 and exp2 must be the same type or must be convertable to the same type by Pascal's default conversion rules (for example, integer converted to a real).

If exp1 and exp2 are unsigned scalars or pointers, DOMAIN Pascal performs an unsigned comparison. (The scalar data types are integers, Boolean, character, and enumerated.) If they are **real**, **single**, or **double**, a floating-point comparison is done, while if they are signed integers, DOMAIN Pascal performs a signed comparison.

See also **max**.

EXAMPLE

```
program min_example;

var
    storenum : integer;
    lowprice, x, y, newprice1, newprice2 : real;

begin
{ The program finds the lowest discounted price for an item that two }
{ stores sell. The stores have different regular prices and are      }
{ featuring different discount rates: 18% at the first, and 15% at   }
{ the second.                                                        }

write ('Enter the regular price at the first store: ') ;
readln (x);
write ('Enter the regular price at the second store: ');
readln (y);

newprice1 := x - (x * 0.18);
newprice2 := y - (y * 0.15);

lowprice := MIN(newprice1,newprice2);
if lowprice = newprice1 then
    storenum := 1
else
    storenum := 2;

write ('The best discounted price is at store number ', storenum:1);
writeln (' and it is ', lowprice:6:2);
end.
```

Using This Example

Following is a sample run of the program named min_example:

```
Enter the regular price at the first store: 1599.99
Enter the regular price at the second store: 1515.15
The best discounted price is at store number 2 and it is 1287.88
```

Mod -- Calculates the remainder upon division of two integers.

FORMAT

d1 **mod** d2 {**mod** is an operator.}

Arguments

d1, d2	Any integer expressions.

Operator Returns

The **mod** operator returns an integer value.

DESCRIPTION

DOMAIN Pascal's **mod** operator works like standard Pascal's **mod** operator when d1 is positive. When d1 is negative and you compile without the **-iso** switch, DOMAIN Pascal's **mod** operator works in a nonstandard manner.

When d1 is Positive

The expression `(d1 MOD d2)` produces the remainder of d1 divided by d2. Therefore, the expression `(d1 MOD d2)` always evaluates to an integer from 0 up to, but not including, |d2|. For example, consider the following results:

```
 9 MOD  3 is equal to 0
10 MOD  3 is equal to 1
11 MOD  3 is equal to 2
12 MOD  3 is equal to 0
13 MOD  3 is equal to 1
13 MOD -3 is equal to 1
```

(To find the quotient (i.e., nonfractional) portion of the division, use the **div** operator described earlier in this encyclopedia.)

When d1 is Negative

If d1 is negative, then `(d1 MOD d2)` equals

```
-1 * remainder of |d1| divided by |d2|
```

For example, consider the following results:

```
 -9 MOD -3 is equal to  0           -9 MOD 3 is equal to  0
-10 MOD -3 is equal to -1          -10 MOD 3 is equal to -1
-11 MOD -3 is equal to -2          -11 MOD 3 is equal to -2
-12 MOD -3 is equal to  0          -12 MOD 3 is equal to  0
-13 MOD -3 is equal to -1          -13 MOD 3 is equal to -1
```

Compiling With the -iso Switch

If you compile with the **-iso** switch (described in Chapter 6), **mod** follows the standard Pascal rules. That is, **mod** returns a value `result` such that:

```
result := d1 - (d1 DIV d2) * d2
```

Since a negative modulus is illegal under standard Pascal rules, if `result` is negative, then:

```
result := result + d2
```

EXAMPLE

```
PROGRAM mod_example;
{This program uses the MOD function to find the coming leap years.}

CONST
    cycle = 4;

VAR
    remainder, year : integer16;

BEGIN
   for year := 1985 to 1999 do
      begin
         remainder := year MOD cycle;
         if remainder = 0
            then writeln(year:4, ' is a leap year');
      end;
END.
```

Using This Example

If you execute the sample program named mod_example, you get the following output:

```
1988 is a leap year
1992 is a leap year
1996 is a leap year
```

New -- Allocates space for storing a dynamic variable.

FORMAT

new(p)	{Short form. **New** is a procedure.}
new(p, tag1, . . . ***tagN*****)**	{Long form.}

Arguments

tag	An input argument that names one or more constants. Tag is valid only if p^ is a record. The maximum number of tags is the number of tag fields in the record to which p points.
p	A pointer variable used for input and output. Pascal creates a dynamic variable of the type to which p points. After allocating this variable, Pascal returns the address of the newly allocated dynamic variable into p. The contents of the address pointed to is undefined. If there was insufficient address space or disk space remaining to satisfy the request for dynamic memory, then DOMAIN Pascal returns the value **nil** in p.

DESCRIPTION

New causes Pascal to allocate enough space for storing one occurrence of a dynamic variable. You use **new** to create dynamic space and **dispose** (described earlier in this encyclopedia) to deallocate the dynamic space.

You can use the short form of **new** to allocate any kind of dynamic variable. The long form of **new** is only useful for allocating dynamic variant records.

The Short Form

Consider the following record declaration:

```
TYPE
   employeepointer = ^employee;
   employee = record
      first_name : array[1..10] of char;
      last_name  : array[1..14] of char;
      next_emp   : employeepointer;
   end;

VAR
   current_employee : employeepointer;
```

If you want to store employee records dynamically, then you must call `NEW(current_employee)` for every occurrence of an employee. To allocate space for 100 employees, call `NEW(current_employee)` 100 times. You can assign values to an employee record only after Pascal has allocated space for an occurrence.

The Long Form

Pascal uses tag1..*tagN* to help determine the amount of space to allocate for a variant record. Tag1..*tagN* corresponds to the tag fields of the variant record. For example, consider the type declaration for the following variant record:

```
TYPE
   emp_stat      = (exempt, nonexempt);
   workerpointer = ^worker;
   worker = record
      first_name : array[1..10] of char;
      last_name  : array[1..14] of char;
      next_emp   : workerpointer;
      CASE emp_stat OF
           exempt    : (salary   : integer16);
           nonexempt : (wages    : single;
                        plant    : array[1..20] of char);
   end;

VAR
   current_worker : workerpointer;
```

Because worker contains a tag field, you have the option of passing the value of a constant to **new**; for example:

```
NEW(current_worker, exempt)
```

Since tag1 is exempt, when DOMAIN Pascal allocates space for one worker record, it allocates two bytes for the variant portion (since **integer16** takes up only two bytes). If tag1 had been nonexempt, DOMAIN Pascal would have allocated the space necessary (24 bytes) to hold it.

Note that the number of constants you pass to **new** must be less than or equal to the number of tag fields in the record declaration.

For machines with a larger address space (DNx60 machines, DN3000, etc.), you can access that larger amount of space by performing the following two steps:

1. Use **new** to allocate a large amount of memory (such as a megabyte) at the beginning of the program's execution.

2. Immediately after allocating the memory, use **dispose** to deallocate it.

These steps cause the operating system to increase the number of memory pages it allocates to your program. You should only use this technique if it is possible that your program may run out of address space.

EXAMPLE

```
Program build_a_linked_list;
{ The following example uses NEW and DISPOSE to create and disassemble a }
{ linked list.  For a description of the theory of linked lists, consult }
{ a Pascal tutorial. }

TYPE
   studentpointer = ^ student;
   student = record
      name : array[1..30] of char;
      age  : integer16;
      next_student : studentpointer;
   end;

VAR
   base     : studentpointer;
   a_name   : array[1..30] of char;
   an_age   : integer16;
   option   : char;
   done     : boolean;


Procedure print_list; { Print the linked list in order. }
VAR
   ns : studentpointer;
BEGIN
    ns := base;
    writeln;
    while ns <> NIL do
with NS^ do
         begin
            writeln(name, ' - ', age);
            ns := next_student;
         end;
END;

Procedure Enter_data;
VAR
   ns, previous : studentpointer;
BEGIN
   base := nil;

   repeat
       write('Enter the name of a student (or end to stop) -- ');
       readln(a_name);
       if a_name  'end' then
          begin
             NEW(ns);     { Allocate space for a new occurrence of a student.}

             write('Enter his or her age -- ');  readln(an_age);

             if base = nil
                then base := ns                   {Set base to first record.}
                else previous^.next_student := ns; {Add record to end of list}

{ Initialize fields of new record. }
             ns^.name := a_name;
            ns^.age  := an_age;
            ns^.next_student := nil;

            previous := ns;     { Save pointer to this new student.}
         end
   until a_name = 'end';
END;

Procedure delete_a_student;
VAR
   ns, previous : studentpointer;
BEGIN
   previous  := base;
   ns := base;

   write('What is the name of the student you want to delete? -- ');
   readln(a_name);

   while ns <> nil do
      begin
         if ns ^.name = a_name then   {Delete record.}
            begin
               if ns = base then
                  base := ns^.next_student
               else
                  previous^.next_student := ns^.next_student;
               DISPOSE(ns);
               exit;
            end
         else
            begin
               previous := ns;
               ns := ns^.next_student;
            end; {if}
      end; {while}
END; {delete_a_student}

BEGIN {main}
   base := nil;
   enter_data;
   print_list;

   repeat
      if base = nil then return;
      write('Do you want to delete a student from the list? (y or n) -- ');
      readln(option);
      done := not (option in ['y', 'Y']);
      if not done then
         begin
            delete_a_student;
            print_list;
         end
   until done;
END.
```

Using This Example

Following is a sample run of the program named build_a_linked_list:

```
Enter the name of a student (or end to stop) -- Kerry
Enter his or her age -- 28
Enter the name of a student (or end to stop) -- Jan
Enter his or her age -- 27
Enter the name of a student (or end to stop) -- Lance
Enter his or her age -- 29
Enter the name of a student (or end to stop) -- end

Kerry                           -         28
Jan                             -         27
Lance                           -         29
Do you want to delete a student from the list? (y or n) -- y
What is the name of the student you want to delete? -- Jan

Kerry                           -         28
Lance                           -         29
Do you want to delete a student from the list? (y or n) -- n
```

Next -- Jump to the next iteration of a For, While, or Repeat loop. (Extension)

FORMAT

Next is a statement that neither takes arguments nor returns values.

DESCRIPTION

You use **next** to skip over the *current iteration* of a loop. You can only use **next** within a **for**, **while**, or **repeat** loop. If **next** appears elsewhere in a program, DOMAIN Pascal issues an error. **Next** tells DOMAIN Pascal to ignore the remainder of the statements within the body of the loop for one iteration. For instance, consider the following example:

```
FOR x := 5 to 35 do
   begin
        . . .
        if (x MOD 10) = 0 then NEXT;
        . . .
   end;
```

When x is 10, 20, or 30, DOMAIN Pascal ignores the statements following the **next**. You can use a **goto** statement instead of a **next** statement. For example, you can rewrite the preceding example to look like the following:

```
FOR x := 5 to 35 do
   begin
        . . .
        if (x MOD 10) = 0 then GOTO 100;
        . . .
100:    end;
```

EXAMPLE

```
PROGRAM next_example;
{ This program counts the occurrences of the digits 0 through 9 in a line }
{ of integers and real numbers.                                           }

CONST
   blank = ' ';
   decimal_point = '.';

VAR
   dig : char;
   count_digits : array[48..57] of integer16 := [* of 0];
   x  : integer16;

BEGIN
   writeln('Enter a line of integers and real numbers:');
   repeat
      read(dig);
      if ((dig = blank) or (dig = decimal_point)) then NEXT;
      write(dig, ' ');
      count_digits[ord(dig)] := count_digits[ord(dig)] + 1;
   until eoln;
   writeln;
   for x := 48 to 57 do
   writeln( count_digits[x]:2, ' of the digits were ',  chr(x):1, 's');
END.
```

Using This Example

Following is a sample run of the program named next_example:

```
Enter a line of integers and/or real numbers.
1741374.13 33 821
1 7 4 1 3 7 4 1 3 3 8
 0 of the digits were 0s
 4 of the digits were 1s
 1 of the digits were 2s
 4 of the digits were 3s
 3 of the digits were 4s
 0 of the digits were 5s
 0 of the digits were 6s
 2 of the digits were 7s
 1 of the digits were 8s
 0 of the digits were 9s
```

Nil -- A special pointer value that points to nothing.

FORMAT

Nil is a reserved word. You can only use it in expressions. **Nil** is a pointer value; therefore, you must assign it to or compare it to a pointer variable. **Nil** is guaranteed never to point to an object.

DESCRIPTION

Use **nil** when you must assign a value to a pointer, but you don't know what that value should be. For example, when creating a linked list, you can set the last record in the list to point to **nil**. Then, when walking through the list, you can easily find the end of the list by checking for **nil**.

EXAMPLE

For a sample program that uses **nil**, refer to the listing for **new** earlier in this chapter.

Not -- Returns true if an expression evaluates to false.

FORMAT

not b {**not** is a unary operator.}

Arguments

b	Any Boolean expression.

Operator Returns

The result of a **not** operation is always a Boolean value.

DESCRIPTION

If b evaluates to false, then **not** b evaluates to true. If b evaluates to true, then **not** b evaluates to false.

Note that you can put **and** or **or** immediately before **not**. Note the order of precedence in an expression like the following:

```
a AND NOT b          actually means          a AND (NOT b)
```

Another potentially confusing expression is the following:

```
NOT a AND b          actually means          (NOT a) AND b
```

Please refer to the order of precedence rules at the beginning of this chapter.

EXAMPLE

```
PROGRAM not_example;
VAR
   pet_lover, timid : boolean;

BEGIN
   writeln('Career aptitude test.', chr(10));
   write('You like pets (true or false) -- '); readln(pet_lover);
   write('You are timid (true or false) -- '); readln(timid);

   if pet_lover and NOT timid
      then writeln('Have you considered becoming a lion tamer?')
      else writeln('Plastics...there''s a great future in plastics.');
END.
```

Using This Example

This program is available on-line and is named not_example.

Odd -- Tests whether the specified integer is an odd number.

FORMAT

odd(i) {**odd** is a function.}

Argument

i	Any integer expression.

Function Returns

The **odd** function returns a Boolean value.

DESCRIPTION

Odd returns true if i is an odd integer and false if i is an even integer.

EXAMPLE

```
PROGRAM odd_example;

VAR
   i : integer;

BEGIN
   write('Enter an integer -- ');
   readln(i);

   if ODD(i)
      then writeln(i:1, ' is an odd number.')
      else writeln(i:1, ' is an even number.');
END.
```

Using This Example

Following is a sample run of the program named odd_example:

```
Enter an integer -- 14
14 is an even number.
```

Of -- Refer to Case earlier in this encyclopedia.

Open -- Opens a file so that you can eventually read from or write to it. (Extension)

FORMAT

open(file_variable, pathname, file_history, *error_status, buffer_size*); {**open** is a procedure.}

Arguments

file_variable — A variable having the **text** or **file** data type.

pathname — The name of the file that you want to open. Pathname is a string constant or string variable, that you specify in any of the following five ways:

- Enter a DOMAIN pathname as defined in *Getting Started With Your DOMAIN System.*
- Enter a string in the form '^n', where n is an integer from 1 to 9. N corresponds to the ordinal value of the arguments that the user passes to the program when he or she executes or debugs the program. For example, suppose you compile DOMAIN Pascal source code to create executable object file `sample.bin`. You can pass the two arguments xxx and yyy by executing the program as follows:

  ```
  $ sample.bin xxx yyy
  ```

  The preceding command line causes DOMAIN Pascal to assign xxx to '^1' and yyy to '^2'.
- Enter a string in the form '*prompt-string'. At runtime, DOMAIN Pascal prints the prompt-string at standard output, and then reads the user's response from standard input. (The response should be the name of the file to be opened.) The prompt-string can contain any printable character except blanks; DOMAIN Pascal stops printing at the first blank it encounters. An asterisk by itself tells DOMAIN Pascal to read the response from standard input without printing a prompt at standard output.
- A string in the form '-STDIN' or '-STDOUT'. These strings correspond to the streams that the operating system opens automatically. However, specifying one of these strings does not cause an error. (See Chapter 8 for an explanation of streams.)
- A variable or constant containing any of the preceding items.

file_history — A variable or string that tells the **open** procedure how to open the file. The variable or string must have one of the following three values:

- 'NEW' -- If the file exists, DOMAIN Pascal reports an error. If the file does not exist, DOMAIN Pascal creates the file and then opens it.
- 'OLD' -- If the file exists, DOMAIN Pascal opens it. If the file does not exist, DOMAIN Pascal reports an error.

- 'UNKNOWN' -- If the file exists, DOMAIN Pascal opens it. If the file does not exist, DOMAIN Pascal creates the file and then opens it.

Remember to enclose the file_history within apostrophes (e.g., 'NEW').

error_status An optional argument. If you specify an *error_status*, it must have an **integer32** data type. At runtime, DOMAIN Pascal returns a hexadecimal number into *error_status* which has the following meaning:

```
0 -- no error or warning occurred.

greater than 0 -- an error occurred.

less than 0 -- a warning occurred.
```

Your program is responsible for handling any errors. We detail error handling in Chapter 9.

buffer_size An optional argument that may only be specified for files of type **text**. *Buffer_size* must be at least as long as the longest line in the file being read; if it is shorter, the excess characters in a line are truncated. If the file is open for writing only, you don't need to specify a large *buffer_size*. No data is lost even if a line being written is longer than *buffer_size*. The default size is 256 bytes.

DESCRIPTION

Before you can read from or write to a file, you must first open it for I/O operations. To open a permanent file, you must use the **open** procedure. To open a temporary file, use the **rewrite** procedure without using an **open** procedure.

After you've opened a file, you then specify whether it is available for reading (by calling **reset**) or for writing (by calling **rewrite**). Note that you do not need to open the standard input (**input**) and standard output (**output**) files before attempting to read from or write to them. They are always open.

When your program terminates, the operating system automatically closes all opened files; however, please refer to the description of the **close** procedure earlier in this chapter.

For a complete overview of DOMAIN I/O, see Chapter 8.

EXAMPLE

```
Program open_example;

{ This program uses a variety of techniques to open three files.       }
{ In order for it to work properly, you must pass the pathname of a    }
{ file as the first argument on the execution or debug command line.   }
{ For example, if you compile this program to create open_example.bin, }
{ then you could invoke the program with the following command:        }
{      $ open_example.bin //arnie/nouveau/comps                        }

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%LIST;

CONST
    name_of_file = 'annabel_lee';
    file3        = '*filename--';
```

```
VAR
    poem, paragraph, stanza  : text;
    statrec                  : status_$t;

BEGIN

{ Open an existing file.}
     OPEN(poem, name_of_file, 'OLD', statrec.all);
{ If there was no error on open, then specify that the file be    }
{ open for reading.                                               }
     if statrec.all = 0
        then reset(poem)
        else writeln('Difficulty opening ', name_of_file);



{ Open a new file.  The pathname of the new file will be the first }
{ argument that you pass on the execution or debug command line.   }
    OPEN(paragraph, '^1', 'NEW', statrec.all);
{ If there was no error on open, then specify that the file be open}
{ for writing.  If there was an error, print the error code.       }
    if statrec.all = status_$ok
        then rewrite(paragraph)
        else writeln('Got error code ', statrec.all, ' on open.');



{ Open a file that may or may not exist. Prompt user for name of           }
{ file at runtime.                                                         }
    OPEN(stanza, file3, 'UNKNOWN', statrec.all);
{ A slightly more sophisticated method of error reporting is to use        }
{ the system call ERROR_$PRINT to print the error message.                 }
{ NOTE: In order to call ERROR_$PRINT, you must specify both               }
{       /sys/ins/base.ins.pas and /sys/ins/error.ins.pas as %INCLUDE files.}
    if statrec.all = status_$ok
       then rewrite(stanza)
       else ERROR_$PRINT(statrec);
END.
```

Using This Example

This program is available on-line and is named open_example.

Or -- Calculates the logical or of two Boolean arguments.

FORMAT

x **or** y {**or** is an operator.}

Arguments

x, y	Any Boolean expressions.

Operator Returns

The result of an **or** operation is always a Boolean value.

DESCRIPTION

Use **or** to find the logical or of expressions x and y. Here is the truth table for **or**:

x	y	**Result**
true	true	true
true	false	true
false	true	true
false	false	false

Compare **or** to **and** and **not** (which are described elsewhere in this encyclopedia). You should also see the description of **or else**, which is described in the **if** listing earlier in this encyclopedia.

NOTE: Some programmers confuse **or** with the exclamation point (!) operator. ! is a bit operator; it causes DOMAIN Pascal to perform a logical or on all the bits in its two arguments. For example, compare the following results:

```
(true OR false) is equal to true
(75 ! 15) is equal to 79
```

Refer to the "Bit Operators" listing earlier in this encyclopedia.

EXAMPLE

```
PROGRAM or_example;

VAR
   tall, good_jumper, good_athlete  : boolean;

BEGIN
   writeln('Career aptitude test.', chr(10));

   write('You are taller than 1.95 meters (true or false) -- '); readln(tall);
   write('You can jump high (true or false) -- '); readln(good_jumper);
   write('You are athletic (true or false) -- '); readln(good_athlete);

   if (tall OR good_jumper) AND good_athlete
      then writeln(chr(10), 'Have you considered playing pro basketball.')
      else writeln(chr(10), 'Computers are a stable field.');
END.
```

Using This Example

This program is available on-line and is named or_example.

Ord -- Returns the ordinal value of a specified integer, Boolean, char, or enumerated expression.

FORMAT

ord(x) {**ord** is a function.}

Argument

x	Any scalar (i.e., integer, Boolean, char, or enumerated) expression.

Function Returns

The **ord** function returns an integer value.

DESCRIPTION

The **ord** function returns the ordinal value of x according to the following rules:

Data Type of x	**Ord Returns**
Integer	Numerical value of x.
Boolean	0 if x is false and 1 if x is true.
Char	x's ASCII value. Appendix B contains an ASCII table.
Enumerated	An integer representing x's position within the enumeration declaration. For instance, in program `ord_example` below, `ORD(rice)` returns 0, `ORD(tofu)` returns 1, and so on up until `ORD(tamari)`, which returns 4.

Note that the **chr** function is the inverse of **ord**.

EXAMPLE

```
PROGRAM ord_example;

TYPE
   macro = (rice, tofu, seaweed, miso, tamari);

VAR
   c : char;
   e : macro;

BEGIN
   c := 'd';
   WRITELN('The ordinal value of ', c, ' is ', ORD(c):3);

   e := seaweed;
   WRITELN('The ordinal value of ', e:7, ' is ', ORD(e):1);
END.
```

Using This Example

If you execute the sample program named ord_example, you get the following output:

```
The ordinal value of d is 100
The ordinal value of SEAWEED is 2
```

Pack -- Copies an unpacked array to a packed array.

FORMAT

pack(unpacked_array, index, packed_array)   {**pack** is a procedure.}

Arguments

unpacked_array	An array that has been defined without the keyword **packed.**
index	A variable that is the same type as the array bounds (**integer**, **boolean**, **char**, or enumerated) of unpacked_array. Index designates the array element in unpacked_array from which **pack** should begin copying.
packed_array	An array that has been defined using the keyword **packed.**

DESCRIPTION

Pack copies an unpacked array to a packed one. However, data access with unpacked arrays generally is faster since data elements are always aligned on word boundaries.

Unpacked_array and packed_array must be of the same type, and for every element in packed_array, there must be an element in unpacked_array. That is, if you have the following **type** definitions

```
TYPE
    x : array[i..j] of single;
    y : packed array[m..n] of single;
```

the subscripts must meet these requirements:

```
j - index >= n - m            {"index" as set in the call to pack}
```

For example, it is legal to use **pack** on two arrays defined like this:

```
TYPE
   big_array   : array[1..100] of integer;
   small_array : packed array[1..10] of integer;
VAR
   grande : big_array;
   petite : small_array;
```

You use index to indicate the array element in unpacked_array from which **pack** should begin copying. For instance, given the previous variable declarations and assuming variable i is an **integer**, this fragment

```
i := 1;
pack(grande, i, petite);
```

tells **pack** to begin copying at grande[1]. **Pack** keeps copying until it reaches the highest index value that petite can take -- which in this case is 10. The remaining elements in grande are not copied.

Index can take a value outside of packed_array's defined subscripts. That is, if in the example above, i equals 50, **pack** copies these values this way:

```
petite[1] := grande[50];
petite[2] := grande[51];
          .
          .
          .
petite[10] := grande[59];
```

See the listing for **unpack** later in this encyclopedia.

EXAMPLE

```
PROGRAM pack_example;

TYPE
    uarray = array[1..50] of integer16;
    parray = packed array[1..10] of integer16;

VAR
    full_range : uarray;
    sub_range  : parray;
    i, j       : integer16;

BEGIN
for i := 1 to 50 do
    full_range[i] := i;

j := 20;
PACK(full_range, j, sub_range);

writeln ('The packed array now contains: ');
for i := 1 to 10 do
    writeln ('sub_range[', i:2, '] = ', sub_range[i]:2);

END.
```

Using This Example

If you execute the sample program named pack_example, you get the following output:

```
The packed array now contains:
sub_range[ 1] = 20
sub_range[ 2] = 21
sub_range[ 3] = 22
sub_range[ 4] = 23
sub_range[ 5] = 24
sub_range[ 6] = 25
sub_range[ 7] = 26
sub_range[ 8] = 27
sub_range[ 9] = 28
sub_range[10] = 29
```

Page -- Insert a formfeed (page advance) into a file.

FORMAT

page(*f*) {**page** is a procedure.}

Argument

f	A **text** variable. *F* is optional. If you do not specify *f*, **page** assumes that the file is standard output (**output**).

DESCRIPTION

Use the **page** procedure to insert a formfeed (ASCII character 12) into the file specified by *f*. **Page** is useful for formatting text that will be printed or for text that meets fixed–length window dimensions. If you print the file on a line printer, the printer advances to the next page when it encounters the formfeed.

Before calling **page**, you must open the file named in *f* for writing. See Chapter 8 for a description of opening files.

EXAMPLE

```
PROGRAM page_example;
{This program demonstrates the PAGE procedure.}

CONST
    lines_in_a_page  =  54;  {Our printer prints 54 lines to a page.}

VAR
    information    : text;
    statint        : integer32;
    x              : integer16;

BEGIN

{ Create a file and open it for writing; exit on error. }
    open(information, 'square_root_table', 'NEW', statint);
    if statint = 0 then
        rewrite(information)
    else
        begin
        writeln('Pascal reports error', statint, ' on OPEN.');
        return;
        end;

{Print the square roots from 1 to 200, inserting }
{formfeeds where needed.                         }

    for x := 1 to 200 do
        begin
        writeln(information, 'The square root of ', x:3, ' is ', sqrt(x));
        if ((x mod lines_in_a_page) = 0) then
            PAGE(information);
        end;

END.
```

Using This Example

This program is available on-line and is named page_example.

Pointer Operations

Chapter 3 explains how to declare pointer types. Here, we describe how to use pointers in the action part of your program.

DESCRIPTION

You can do the following things with a pointer variable:

- Use the **addr** function to assign the virtual address of a variable to the pointer variable.
- Compare or assign the value of one pointer variable to another compatible pointer variable.
- De-reference a pointer variable. De-referencing means that you find the contents of the variable to which the pointer variable was pointing.

The following program fragment does all three things:

```
Program test;

TYPE
   pi = ^integer16;

VAR
   p1quart, p2quart : pi;
   quart1,   quart2 : integer16 := 5;

BEGIN
   p1quart := addr(quart1);
   p2quart := p1quart;
   quart2  := p2quart^;
END.
```

Manipulating Virtual Addresses -- Extension

DOMAIN Pascal supports type transfer functions that are quite useful in manipulating virtual addresses. For example, you cannot directly write a pointer value to a **text** file; however, you can use a type transfer function to transfer the address to an **integer32** value (which can be written). For example, compare the right and wrong ways to write the virtual address of `quart1` to **output**:

```
writeln(p1quart);            {wrong}
writeln(integer32(p1quart)); {right}
```

Invoking Procedure and Function Pointers -- Extension

You de-reference a procedure or function pointer like any other pointer; that is, with the up-arrow (^). In this way, functions can return pointers to other functions; for example:

```
TYPE
    retbool = ^function : boolean;
    retfunptr = ^function : retbool;

VAR
    xp : retbool;
    rf : retfunptr;
    flag : boolean;

FUNCTION myfunc; retbool;
          .
          .
          .

    rf := ADDR(myfunc);
    xp := rf^;
    flag := xp^;
```

The expression `rf^` invokes the `myfunc` function, which returns a pointer to a function that returns a Boolean value. You cannot use the following assignment

```
flag := rf^^;
```

because you cannot de-reference the return value of a function call.

Addressing Procedure and Function Pointers -- Extension

To obtain procedure and function addresses, use the predeclared function **addr**. Thus, there is no ambiguity about a function reference, especially one with no parameters. It is either invoked by name only, or its address is taken by the **addr** function.

Although the **addr** function has been declared to return a **univ_ptr**, the compiler adds extra type checking whenever you try to pass a procedure or function address to a specific procedure or function pointer. In the assignment `pptr := addr(proc2)` from the following program fragment, the declaration for `proc2` must exactly match the template for the procedure type of `pptr`. If not, the compiler reports an error. If, however, the assignment is to a **univ_ptr**, like `xxx := addr(func1)`, the compiler cannot do this extra type checking.

```
VAR
    xxx : UNIV_PTR;
   pptr : ^Procedure(IN i, j : integer;
                     OUT   a : char;
                     VAR   r : real); EXTERN;
BEGIN
   . . .
   xxx := func1;         {This is a call.}
   . . .
   xxx := addr(func1);   {This takes the address.}
   . . .
   pptr := addr(proc2);  {This takes the address and checks.}
```

Pred -- Returns the predecessor of a specified ordinal value.

FORMAT

pred(x) {**pred** is a function.}

Argument

x	An integer, Boolean, char, or enumerated expression.

Function Returns

The **pred** function returns a value having the same data type as x.

DESCRIPTION

Pred returns the predecessor of x according to the following rules:

Data Type of x	Pred Returns
Integer	The numerical value equal to x – 1.
Boolean	False -- even if x already equals false.
Char	The character with the ASCII value one less than the ASCII value of x. If this character (x–1) does not exist, DOMAIN Pascal cannot detect the error.
Enumerated	The identifier to the left of x in the type declaration. If x is the left-most identifier, **pred**'s return value is undefined.

`PRED(FIRSTOF(x))` generally is undefined; however, DOMAIN Pascal does not report an error. DOMAIN Pascal also doesn't report an error if you specify an integer value that is outside the range of the specified integer type. Therefore, your program should test for an out-of-bounds condition.

Compare the **pred** function to the **succ** function.

EXAMPLE

```
PROGRAM pred_example;
TYPE
   jours = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche);

VAR
   i       : integer;
   c1, c2  : char;
   semaine : jours;

BEGIN
   i := 53;  c1 := 'n';  semaine := vendredi;
   writeln('The predecessor of ', i:2, ' is ', pred(i):2);
   c2 := pred(c1);
   writeln('The predecessor of ', c1:1, ' is ', c2);
   writeln('The predecessor of ', semaine:8, ' is ', pred(semaine):8);
END.
```

Using This Example

If you execute the sample program named pred_example, you get the following output:

```
The predecessor of 53 is 52
The predecessor of n is m
The predecessor of VENDREDI is    JEUDI
```

Put -- Writes to a file.

FORMAT

put(f) {**put** is a procedure.}

Argument

f	A variable having the **file** or **text** data type.

DESCRIPTION

If f is a **file** variable, then **put**(f) appends one record to the file symbolized by f. If f is a **text** variable, then **put**(f) appends one character to the file symbolized by f.

Before calling **put**(f), you must assign a record or character to f^. So the sequence for writing out data looks like the following:

```
f^ := record_or_character;
PUT(f);
```

For example, the following program fragment demonstrates output via the **put** procedure:

```
VAR
    primes   : file of integer16;
    poem     : text;
    a_number : integer16;
    a_letter : char;

BEGIN
    .
    .
    .
    a_number := 17;
    primes^  := a_number;
    PUT(primes);    { Append 17 to the file symbolized by primes. }
    . . .

    a_letter := 'Q';
    poem^    := a_letter;
    PUT(poem);      { Append 'Q' to the file symbolized by poem. }
    .
    .
    .
```

Note that the three statements

```
a_letter := ´Q´;
poem^    := a_letter;
PUT(poem);       { Append ´Q´ to the file symbolized by poem. }
```

are identical to the two statements

```
a_letter := ´Q´;
write(poem, a_letter);
```

You must open f for writing (with **rewrite**) before calling **put**.

> NOTE: When you want to close the file on which you were performing **puts**, your program should issue a **writeln** to the file just before closing it. This is in order to flush the file's internal output buffer. If you don't include the **writeln**, the last line of the file may not be written.

EXAMPLE

```
PROGRAM put_example;

{ This program builds a file of student records in file 'his101'.  }

CONST
    file_to_write_to = 'his101';

TYPE
    student =
       record
            name   : array[1..12] of char;
            age    : integer16;
       end;

VAR
    class              : FILE OF student;
    a_student          : student;
    iostat             : integer32;

BEGIN
{ Opens file his101 for writing. }
    open(class, file_to_write_to, 'NEW', iostat);
    if iostat = 0
        then rewrite(class)
        else return;

    repeat
{ Prompt users for input.                              }
       writeln;
       write('Enter the name of a student -- ');
       readln(a_student.name);
       if a_student.name = 'end' then exit;
       write('Enter the age of this student -- ');
       readln(a_student.age);

{ Append each record to the end of the rec file.       }
       class^ := a_student;
       PUT(class);
    until false;
END.
```

Using This Example

This program is available on-line and is named put_example.

Read, Readln -- Reads information from the specified file (or from the keyboard) into the specified variable(s).

FORMAT

read(*f*, var1, ..., *varN*);	{**read** is a procedure.}
and	
readln(*f*, var1, ..., *varN*);	{**readln** is a procedure.}

Arguments

f	A variable having a file data type. For **read**, *f* can be a **text** or a **file** variable. However, for **readln** *f* must be a **text** variable. *F* is optional. If you do not specify *f*, DOMAIN Pascal reads from standard input (**input**), which is usually the keyboard.
var	One or more variables separated by commas. Var can be any real, integer, char, Boolean, subrange, or enumerated variable. (Boolean and enumerated are extensions to the standard.) Var can also be an array variable (see "Array Operations" earlier in this encyclopedia), an element of an array, or a field of a record variable.

DESCRIPTION

Read and **readln** perform input operations. (Refer to the **get** listing earlier in this encyclopedia.) You use **read** or **readln** to gather one or more pieces of data from *f* and store them into var1 through *varN*. **Read** and **readln** store the first piece of gathered data into var1, the second piece into *var2*, and so on until *varN*.

Before calling **read** or **readln**, you must open the file symbolized by *f* for reading. Chapter 8 explains how to do that.

There is a subtle, but important, difference between **read** and **readln**. After a **read**, the stream marker points to the character or component after the last character or component it read from the file. In contrast, after a **readln**, the stream marker points to the character or record after the next end-of-line character in the file. In other words, after getting the data for *varN*, **readln** skips over the remainder of the current line in the input file. (Note that var1 through *varN* may themselves cover several lines of data in the file.) If you call **readln** and var is a record variable, the compiler reports an error; however, it is not an error to call **read** with the same variable -- as long as the record variable is the base type of *f*.

If you call **read** or **readln** when **eof**(*f*) is true, the operating system reports an error.

EXAMPLE

```
PROGRAM read_example;

{ This program demonstrates READLN by reading from the poem stored in      }
{ pathname ´annabel_lee´.                                                   }

CONST
    pathname = ´annabel_lee´;

VAR
    a_line             : string;
    poem               : text;
    title              : array[1..60] of char;
    st                 : integer32;
    count, n           : integer16;

BEGIN

{ Open the file for reading.                                                }
    open(poem, pathname, ´OLD´, st);
    if st = 0
       then reset(poem)
       else begin
               writeln(´Cannot open ´, pathname);
               return;
            end;

    readln(poem, title);
    writeln(´Which line of ´, title);
    write(´do you want to retreive? -- ´);
    readln(n);

    for count := 1 to (n + 1) do
           readln(poem, a_line);
    writeln(output, a_line);

END.
```

Using This Example

Following is a sample run of the program named read_example:

```
Which line of                        Annabel Lee
do you want to retreive? -- 3
That a maiden there lived whom you may know
```

Record Operations

In Chapter 3 you learned how to declare record types and variables. This section explains how to refer to records in the action part of your program.

Referring to Fixed Records

In the action part of your program, you specify a field of a fixed record in the following way:

record_name.field_name

For example, consider the following declaration of a fixed record variable:

```
VAR
   student : record
      n    : array[1..26] of char;
      id   : integer16;
   end;
```

You can assign values to the two fields with the following statements:

```
student.n   := 'Herman Melville';
student.id  := 37;
```

If the data type of the field is itself a record, you must specify the ultimate field in the following way:

record_name.field_name.field_name

For example, consider the following record within a record declaration:

```
TYPE
    name = record
       first  : array[1..10] of char;
       middle : array[1..10] of char;
       last   : array[1..16] of char;
    end;

VAR
    student : record
          n    : name;
          id   : integer16;
    end;
```

You can assign values to all four fields with the following statements:

```
student.n.first  := 'Kerry';
student.n.middle := 'Bruce';
student.n.last   := 'Raduns';
student.id       := 134;
```

Variant Records

In the "Variant Records" section of Chapter 3, the variant records worker and my_code were declared as follows:

```
TYPE
   worker_groups = (exempt, non_exempt);    {enumerated type}

   worker = record     {record type}
              employee : array[1..30] of char; {field in fixed part}
              id_number : integer16;           {field in fixed part}
              CASE wo : worker_groups OF       {variant part}
                 exempt : (yearly_salary   : integer32);
                 non_exempt : (hourly_wage : real);
            end;

   my_code = record
               CASE integer OF            {variant part}
                      1 : (all : array[1..4] of char);
                      2 : (first_half : array[1..2] of char;
                           second_half : array[1..2] of char);
                      3 : (x1  : integer16;
                           x2  : boolean;
                           x3  : char);
                      4 : (rall : single);
             end;

VAR
    w : worker;
   mc : my_code;
```

The following fragment assigns values to w:

```
write('Enter the person''s name -- ');      readln(w.employee);
write('Enter the person''s id number -- '); readln(w.id_number);
write('Enter pay status (exempt or non_exempt) -- '); readln(w.wo);
if w.wo = exempt
   then begin
           write('Enter yearly salary -- '); readln(w.yearly_salary);
        end
   else begin
           write('Enter hourly wage -- '); readln(w.hourly_wage);
        end;
```

NOTE: Suppose you execute the preceding fragment and load values into w.employee, w.id_number, w.wo, and w.hourly_wage. Note that the compiler won't protect you from mistakenly trying to access w.yearly_salary rather than w.hourly_wage.

The following fragment assigns values to mc. (Notice that we do not use the constants 1, 2, 3, and 4 to specify these fields.)

```
write('Enter two characters -- ');  readln(mc.first_half);
write('Enter two more characters -- '); readln(mc.second_half);
writeln('Together, the four characters are ', mc.all);
```

Arrays of Records

A common way to store records is as an array of records. You must use the following format to specify a field in an array of records:

array_name[component].field_name

For example, given the following declaration for `school`:

```
TYPE
   student = record
      age : 11..20;
      class : 7..12;
      name : array[1..20] of char;
   end;

VAR
   school : array[1..1000] of student;
```

you can specify the 500th record as:

```
school[500].age := 15;
school[500].class := 10;
school[500].name := 'John Donne';
```

Repeat/Until -- Executes the statements within a loop until a specified condition is satisfied.

FORMAT

```
repeat                          {repeat is a statement.}
        stmnt; . . .
until cond;
```

Arguments

stmnt	An optional argument. For *stmnt*, specify a simple statement or a compound statement. (Ordinarily, you must bracket a compound statement with a **begin/ end** pair; however, the **begin/end** pair is optional within a **repeat** statement.)
cond	Any Boolean expression.

DESCRIPTION

Repeat marks the start of a loop; **until** marks the end of that loop. At runtime, Pascal executes *stmnt* within that loop until cond is true. As long as cond is false, Pascal continues to execute the statements within the loop.

The following list describes two methods of jumping out of a **repeat** loop prematurely (i.e., before the condition is true):

- Use **exit** to transfer control to the first statement following the **repeat** loop.
- Use **goto** to transfer control outside the loop.

In addition to these measures, you can also execute a **next** statement to skip the remainder of the statements in the loop for one iteration.

EXAMPLE

```
PROGRAM repeat_example;
{ This program demonstrates two different REPEAT loops. }
{ Compare it to while_example.                          }

VAR
   num             : integer16;
   test_completed  : boolean;
   i               : integer32;

BEGIN
    write('Enter an integer -- '); readln(num);

    REPEAT
        num := num + 10;
        writeln(num, sqr(num));
    UNTIL (num > 101);

    writeln;
    test_completed := false;
    REPEAT
        write('Enter another integer (or 0 to stop the program) -- ');
        readln(i);
        if i = 0 then
            test_completed := true
        else
            writeln('The absolute value of ', i:1, ' is ', abs(i):1);
    UNTIL test_completed;

END.
```

Using This Example

Following is a sample run of the program named repeat_example:

```
Enter an integer -- 70
        80       6400
        90       8100
       100      10000
       110      12100

Enter another integer (or 0 to stop the program) -- 4
The absolute value of 4 is 4
Enter another integer (or 0 to stop the program) -- -5
The absolute value of -5 is 5
Enter another integer (or 0 to stop the program) -- 0

{ Now, consider a second run of repeat_example. This time, the user enters }
{ an integer greater than 101. In contrast to while_example, the program   }
{ still will execute the loop once:                                        }

Enter an integer -- 102
       112      12544

Enter another integer (or 0 to stop the program) -- 0
```

Replace -- Substitutes a new record for an existing record in a file. (Extension)

FORMAT

replace(file_variable) {**replace** is a procedure.}

Argument

file_variable	A **file** variable.

DESCRIPTION

Use the **replace** procedure to replace an element in the file specified by file_variable. You can only use **replace** on DOMAIN record-structured (rec) files; you cannot use it to replace an element in an UASC file.

Before calling **replace** you must do the following:

1. Open the file for reading. (Chapter 8 explains how to do this.)
2. Specify the record that you wish to replace. To do this, you can use the **find** procedure (described earlier in this chapter).
3. Store the replacement record by entering a statement of the format: file_variable^ := replacement_record;

The **replace** procedure permits a program to rewrite file components -- for example, to correct errors -- while the file is open for read access. The program need not close the file and reopen it.

NOTE: In this context, "record" means an occurrence in a DOMAIN record-structured file, which may or may not be a DOMAIN Pascal record type. (The occurrence might just as easily be an integer type.)

EXAMPLE

For a full example of **replace**, see the example for the **find** procedure that appears earlier in this encyclopedia.

Reset -- Makes an open file available for reading.

FORMAT

reset(filename)	{**reset** is a procedure.}

Argument

filename	A variable having the **text** or **file** data type.

DESCRIPTION

Before you can read data from a file other than standard input, you must reset the file. If the file is a temporary file and does not already exist, **reset** creates an empty file. If the file is not a temporary file, it must already exist, and you must have previously opened it using **open**. The **open** procedure tells the system to open a file for some type of I/O operation; **reset** tells the system to allow you to read from the file, but prevents you from modifying the file. (See the description of the **find** procedure for one exception to this rule.)

Filename must symbolize an open file.

Calling **reset** sets the stream marker to point to the beginning of the file. Therefore, filename^ will contain the first character or component of the file. You can change the stream marker by reading from the file (with **read, readln,** or **get**) or by calling the **find** procedure.

If the file is empty when you call **reset,** then filename^ is totally undefined. That is, there is no way to predict what the value of filename^ will be.

To open the file for write access you must call **rewrite** instead of **reset.**

EXAMPLE

```
PROGRAM reset_example;
{ Demonstrates reset.  After opening a file (with OPEN), the program }
{ reads the first line of the file and writes it to standard output. }

{ We need the two include files in order to use status_$t and        }
{ error_$print.                                                       }
%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%LIST;

CONST
    pathname_of_file = 'annabel_lee';

VAR
    assignment        : text;
    openstatus        : status_$t;
    a_line            : string;

BEGIN

{Open the file for reading.                                           }
    open(assignment, pathname_of_file, 'OLD', openstatus.all);
    if openstatus.all = status_$ok
       then RESET(assignment)
       else begin
              error_$print(openstatus);  {print any error              }
              return;
            end;
{See Chapter 9 for a discussion of error handling.                    }

{Read the first line of the file.                                     }
    readln(assignment, a_line);

{Write this line to standard output (usually the transcript pad).     }
    writeln(output, a_line);

END.
```

Using This Example

This program is available on-line and is named reset_example.

Return -- Causes program control to jump back to the calling procedure or function. (Extension)

FORMAT

Return is a statement that takes no arguments and returns no values.

DESCRIPTION

Ordinarily, after DOMAIN Pascal executes the last statement in a routine, it returns control to the calling routine. However, you can use **return** to jump back prematurely (i.e., before the last statement) to the calling procedure or function. You can use **return** anywhere in the body of a procedure or function.

Using **return** in the main procedure causes the program to terminate.

EXAMPLE

```
PROGRAM return_example;
{This program demonstrates the RETURN statement. }
VAR
    Ph : single;

Procedure check_Ph;
BEGIN
    if (Ph < 2.0) or (Ph > 14.0) then
        begin
        writeln('You have entered an invalid result.');
        RETURN;
        end
    else if (Ph <= 4.5) then
        writeln('You have entered a valid (but suspicious) result.')
    else     {Ph > 4.5}
        writeln('You have entered a valid result.');

    writeln('Thank you for your cooperation.');
END;

BEGIN  {main procedure}
    write('Enter the Ph of the test sample -- ');  readln(Ph);
    check_Ph;
END.
```

Using This Example

This program is available on-line and is named return_example.

Rewrite -- Makes an open file available for writing only.

FORMAT

rewrite(filename) {**rewrite** is a procedure.}

Argument

filename	A variable having the **text** or **file** data type.

DESCRIPTION

Before you can write data to a permanent file other than standard output, you must do two things. First, you must open the file with the **open** procedure. Second, you must call the **rewrite** procedure. **Open** tells the system to open a file for some type of I/O operations; **rewrite** tells the system to allow you to modify the open file. Filename must symbolize an open file.

To open a temporary file for writing, you merely have to call the **rewrite** procedure (i.e., don't call the **open** procedure).

NOTE: **Rewrite** clears an existing file of its entire contents. To avoid inadvertently erasing an important file, you might consider using the file_history value 'NEW' when you call **open**.

Rewrite sets the stream marker to the beginning of the file. Each call to **write**, **writeln**, or **put** advances the stream marker.

After calling **rewrite**, the file is empty. Therefore, the value of filename^ is totally undefined. That is, there is no way to predict what its value will be.

To open the file for read access you must call **reset** instead of **rewrite**.

EXAMPLE

```
PROGRAM rewrite_example;

{ The program will prompt you for the name of the file to open. }
{ Using OPEN and REWRITE, the program will open a file for writing. }
{ Then, you will be given a chance to write to the file. }

{ We need the two include files in order to use status_$t and error_$print. }
%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%LIST;

VAR
    name_of_file      : array[1..50] of char;
    profound          : text;
    openstatus        : status_$t;
    a_line            : string;

BEGIN

{ Prompt the user for the name of a file to open. }
    write('What is the pathname of the file you want to write to -- ');
    readln(name_of_file);

{ Open the file for writing. }
    open(profound, name_of_file, 'NEW', openstatus.all);
    if openstatus.all = status_$ok
       then REWRITE(profound)
       else begin
              error_$print(openstatus);  { Print an error message. }
              return;
            end;
{ See Chapter 8 for a discussion of error handling. }

{ Prompt the user. }
    writeln('Now enter a line of text.');
    readln(a_line);

{ Write the line out to the open file. }
    writeln(profound, a_line);

END.
```

Using This Example

This program is available on-line and is named rewrite_example.

Round -- Converts a real number to the closest integer.

FORMAT

round(n) {**round** is a function.}

Argument

n	Any real expression.

Function Returns

The **round** function returns an integer value.

DESCRIPTION

The **round** function rounds (up or down) n to the closest integer. If the decimal part of n is equal to or greater than .5, the **round** function rounds up. (Compare **round** to **trunc.**)

EXAMPLE

```
PROGRAM round_example;

VAR
   x : REAL;
   y : INTEGER;

BEGIN
   x := 54.2;   y := ROUND(x);   WRITELN(y);
   x := 54.5;   y := ROUND(x);   WRITELN(y);
   x := 54.8;   y := ROUND(x);   WRITELN(y);
END.
```

Using This Example

If you execute the sample program named round_example, you get the following output:

```
54
55
55
```

Rshft -- Shifts the bits in an integer a specified number of spaces to the right. (Extension)

FORMAT

rshft(num, sh) {**rshft** is a function.}

Arguments

num, sh	Integer expressions.

Function Returns

The **rshft** function returns an integer value.

DESCRIPTION

The **rshft** function shifts the bits in num to the right sh places. **Rshft** does not wrap bits around from the right edge to the left; instead, **rshft** shifts zeros in from the left end.

Rshft does not preserve the sign bit. The sign bit moves to the right just like every other bit. This means that if num is negative and is a 32-bit integer, the result of an **rshft** is always positive. Of course, if num already is positive, the result of an **rshft** will still be positive.

If num is a 16-bit signed integer and if the result of the function is to be stored in a 16-bit integer variable, then **rshft** sign-expands num to a 32-bit integer, performs the shift, and converts it back to a 16-bit integer. The expansion and contraction means that for a 16-bit negative num where sh is less than or equal to 16, **rshft** always returns a negative number.

Consider this example. Suppose num is 16 bits and equals -9. You perform an **rshft** with sh equaling 3 and put the result back in a 16-bit integer. Here's what happens at each step:

```
Before the rshft                                    1111111111110111 = -9
Convert to 32-bit integer           11111111111111111111111111110111
Rshft 3 bits                        00011111111111111111111111111110
Convert back to 16-bit integer                      1111111111111110 = -2
```

If you print the **rshft** result *before* it is converted back to 16 bits, you get the number represented in the third step above which, of course, is a different number than the final result. Write your code like this to get that 32-bit result

```
writeln(rshft(num,3));
```

instead of like this

```
answer := rshft(num,3);             {Assume answer is a 16-bit integer.}
writeln(answer);
```

Results are unpredictable if sh is negative.

Compare **rshft** to **lshft** and **arshft**.

EXAMPLE

See the example shown in the **arshft** listing earlier in this encyclopedia.

Set Operations

In Chapter 3 you learned how to declare set variables. This section explains how to use set variables in the code portion of your program.

ASSIGNMENT

To assign value(s) to a set variable, use one of the following formats:

```
set_variable := [];
set_variable := [el, el, ... el];
set_variable := [el .. el];
set_variable := set_expression  set_operator  set_expression;
```

The brackets are mandatory. El must be an expression with a value having the same data type as the base type of the set variable.

(The set_operators are detailed later in this listing.)

The following program fragment shows seven possible set assignments for the paint set:

```
TYPE
   colors = (white, beige, black, red, blue, yellow, green);

VAR
   c : colors;  {enumerated variable}
   paint1, paint2, paint3, paint4, paint5, paint6, paint7 : SET OF colors;

BEGIN
   c := blue;
   paint1 := [];                {Null set.}
   paint2 := [rojo];            {Illegal assignment.}
   paint3 := [red];
   paint4 := [beige, green, black];
   paint5 := [white .. green];   {All seven elements.}
   paint6 := [beige .. blue];    {beige, black, red, and blue.}
   paint7 := [c];                {blue.}
```

SET OPERATORS

Table 4-10 shows the seven set operators DOMAIN Pascal supports. The following subsections describe these operators individually.

Table 4-10. Set Operators

Set Operator	Operation
+	Union of two sets
*	Intersection of two sets
-	Set exclusion
=	Set equality
<>	Set inequality
<=	Subset
>=	Superset
in	Inclusion

Union

The union of two sets is a set containing all members of both sets. In the following example, `paint3` contains the union of sets `paint1` and `paint2`:

```
TYPE
    colors = (white, beige, black, red, blue, yellow, green);

VAR
    c : colors;
    paint1, paint2, paint3 : SET OF colors := [];

BEGIN
    paint1 := [white, black, red];
    paint2 := [black, yellow];
    paint3 := paint1 + paint2;
{paint3 will contain white, black, red, and yellow.}
```

If there are duplicates (e.g., `black`), the resulting set does not store the duplicate value twice. Thus, `paint3` contains `black` only once.

Intersection

The intersection of two sets is a set containing only the duplicate elements. In the following example, paint3 contains the intersection of sets paint1 and paint2:

```
TYPE
    colors = (white, beige, black, red, blue, yellow, green);

VAR
    c : colors;
    paint1, paint2, paint3 : SET OF colors := [];

BEGIN
    paint1 := [white, black, red];
    paint2 := [black, yellow];
    paint3 := paint1 * paint2;
{paint3 will contain black.}
```

Set Exclusion

Pascal finds the result of a set exclusion operation by starting with all the elements in the left operand and crossing out any of the elements that are duplicated in the right operand. The following program fragment demonstrates two set exclusion operations:

```
TYPE
    colors = (white, beige, black, red, blue, yellow, green);

VAR
    c : colors;
    paint1, paint2, paint3 : SET OF colors := [];

BEGIN
    paint1 := [white, black, red];
    paint2 := [black, yellow];

    paint3 := paint1 - paint2;
{paint3 will contain white and red.}

    paint3 := paint2 - paint1;
{paint3 will contain yellow.}
```

Set Equality and Inequality

The result of a set equality (=) or inequality (<>) operation is a Boolean value. If two set variables contain exactly the same elements (or are both null sets), then = is true and <> is false. The following program fragment demonstrates set equality:

```
TYPE
    colors = (white, beige, black, red, blue, yellow, green);

VAR
    c : colors;
    paint1, paint2 : SET OF colors := [];

BEGIN
    paint1 := [white, black, red];
    paint2 := [black, yellow];
    if paint1 = paint2
        then writeln('The two sets contain the same elements.');
        else writeln('The two sets do not contain the same elements.');
```

Subset

The result of a subset operation (<=) is a Boolean value. If the first operand is a subset of the second operand, then the result is true; otherwise, the result is false.

```
TYPE
    colors = (white, beige, black, red, blue, yellow, green);

VAR
    c : colors;
    paint1, paint2 : SET OF colors := [];

BEGIN
    paint1 := [white, black, red];
    paint2 := [white, red];
    if paint2 <= paint1    {this is true}
        then writeln ('Paint2 is a subset of paint1');
```

Superset

The result of a superset operation (>=) is a Boolean value. If the first operand is a superset of the second operand, then the result is true; otherwise, the result is false.

```
TYPE
    colors = (white, beige, black, red, blue, yellow, green);

VAR
    c : colors;
    paint1, paint2 : SET OF colors := [];

BEGIN
    paint1 := [white, black, red];
    paint2 := [white, red];
    if paint1 >= paint2  {this is true.}
        then writeln('Paint1 is a superset of paint2.');
```

Inclusion

See the separate **in** listing earlier in this encyclopedia.

EXAMPLE

```
PROGRAM set_example;
{This program demonstrates I/O with set variables. You cannot  }
{use a set variable as an argument to any of the predeclared   }
{Pascal I/O procedures, so you must use a somewhat roundabout  }
{method involving the base type of the set.                    }

TYPE
    possible_ingredients =
        (sugar, nuts, chips, milk, flour, carob, salt, bkg_soda);
VAR
    pi      : possible_ingredients;
    cookies : set of possible_ingredients := [];
    answer  : char;

BEGIN
{Read the proper cookie ingredients and store }
{them in the cookies variable.                }
    for pi := sugar to bkg_soda do
        begin
        write('Should the recipe contain ', pi:4, '? (y or n) -- ');
        readln(answer);
        if (answer = 'y') or (answer = 'Y') then
            cookies := cookies + [pi];
        end; {for}

{Write the list of ingredients.               }
    writeln(chr(10), 'The ingredients are: ');
    for pi := sugar to bkg_soda do
        if pi IN cookies then
            writeln(pi);

END.
```

Using This Example

Following is a sample run of the program named set_example:

```
Should the recipe contain SUGAR? (y or n) -- y
Should the recipe contain NUTS? (y or n) -- y
Should the recipe contain CHIPS? (y or n) -- y
Should the recipe contain MILK? (y or n) -- n
Should the recipe contain FLOUR? (y or n) -- y
Should the recipe contain CAROB? (y or n) -- n
Should the recipe contain SALT? (y or n) -- y
Should the recipe contain BKG_SODA? (y or n) -- y

The ingredients are:
          SUGAR
           NUTS
          CHIPS
          FLOUR
           SALT
       BKG_SODA
```

Sin -- Calculates the sine of the specified number.

FORMAT

sin(number) {**sin** is a function.}

Argument

number	Any real or integer expression.

Function Returns

The **sin** function returns a real value (even if number is an integer).

DESCRIPTION

The **sin** function calculates the sine of a number. This function assumes that the argument (number) is a radian measure (as opposed to a degree measure). (Refer also to the **cos** listing earlier in this encyclopedia.)

EXAMPLE

```
PROGRAM sin_example;
{This program demonstrates the SIN function.}

CONST
   pi = 3.1415926535;

VAR
   angle_in_radians, c1, converted_to_radians, c2, angle_in_degrees   : REAL;

BEGIN

   write('Enter an angle in radians -- ');
   readln(angle_in_radians);
   c1 := SIN(angle_in_radians);
   writeln('The sine of ', angle_in_radians:5:3, ' radians is ', c1:5:3);

{The following statements show how to convert from degrees to radians. }
   write('Enter another angle (in degrees) -- ');
   readln(angle_in_degrees);
   converted_to_radians := ((angle_in_degrees * pi) / 180.0);
   c2 := SIN(converted_to_radians);
   writeln('The sine of ', angle_in_degrees:5:3, ' is ', c2:5:3);

END.
```

Using This Example

Following is a sample run of the program named sin_example:

```
Enter an angle in radians -- 1.0
The sine of 1.000 radians is 0.841
Enter another angle (in degrees) -- 14.2
The sine of 14.200 is 0.245
```

Sizeof -- Returns the size (in bytes) of the specified data object. (Extension)

FORMAT

The **sizeof** function has two formats:

sizeof(x)	{first form}
sizeof(x, tag1, . . . *tagN)*	{second form}

Arguments

x	The name of a type (standard or user-defined), a variable, a constant, or a string.
tag	One or more constants corresponding to the fields in a variant record. You specify these tags only if you want to find the size of a variant record. The number of tags can be no greater than the number of tag fields in the variant record.

Function Returns

The function returns an integer value.

DESCRIPTION

Sizeof returns an integer equal to the number of bytes that the program uses to store x.

You must often supply a string and its length as input arguments when calling a procedure or function. You can use **sizeof** to calculate the string's length, although the way you call **sizeof** affects the answer you get. For example, if your code includes the following:

```
VAR
   length : integer;
   animal : string;
           .  .  .
animal := 'wildebeest';
length := SIZEOF(animal);
```

sizeof returns 80 because animal is declared to be a **string**, and **string** is defined as being an array of 80 **chars**. However, if you call the function this way:

```
length := SIZEOF('wildebeest');
```

sizeof returns a value of 10.

To find the size of a specified variant record, you must pass both the name of the variant record type (or variable), and tag fields. For example, consider the following record declaration:

```
TYPE
   worker_stat  = (exempt, nonexempt);
   worker = record
               name : array[1..24] of char;
               case worker_stat of
                 exempt    : (salary  : integer16);
                 nonexempt : (wages   : single;
                              plant   : array[1..20] of char);
               end;
   end;
```

To find the size of a record if worker_stat equals exempt, call **sizeof** as follows:

```
SIZEOF(worker_stat, exempt)
```

To find the size of a record if worker_stat equals nonexempt, call **sizeof** as follows:

```
SIZEOF(worker_stat, nonexempt)
```

EXAMPLE

```
PROGRAM sizeof_example;
{ This program demonstrates the SIZEOF function. }

CONST
    tree   = 'ficus';
TYPE
    student = RECORD
        name : array[1..19] of char;
        age  : integer16;
        id   : integer32;
    end;
VAR
    t2    : integer16;

BEGIN

    writeln('The size of constant tree is ', SIZEOF(tree):1);
    writeln('The size of variable t2 is ', SIZEOF(t2):1);
    writeln('The size of an integer32 variable is ', SIZEOF(integer32):1);
    writeln('The size of a student record is ', SIZEOF(student):1);

END.
```

Using This Example

If you execute the sample program named sizeof_example, you get the following output:

```
The size of constant tree is 5
The size of variable t2 is 2
The size of an integer32 variable is 4
The size of the student record type is 26
```

Sqr -- Calculates the square of a specified number.

FORMAT

sqr(n) {**sqr** is a function.}

Argument

n	Any integer or real expression.

Function Returns

The **sqr** function returns an integer if n is an integer, and returns a real number if n is real.

DESCRIPTION

The **sqr** function calculates n * n. A potential problem for users is that the square of a large **integer16** value often exceeds the maximum value (32,767) for **integer16** variables. If this error is possible in your program, assign the square of an **integer16** variable to an **integer32** variable.

EXAMPLE

```
PROGRAM sqr_example;

VAR
   i_short  : integer16;
   i_long   : integer32;
   r1, r2  : real;

BEGIN
   write('Enter an integer -- '); readln(i_short);
   i_long := SQR(i_short);
   writeln('The square of ', i_short:1, ' is ', i_long:1);

   write('Enter a real number -- '); readln(r1);
   r2 := SQR(r1);
   writeln('The square of ', r1:1, ' is ', r2:1);
END.
```

Using This Example

Following is a sample run of the program named sqr_example:

```
Enter an integer -- 1100
The square of 1100 is 1210000
Enter a real number -- -5.23
The square of -5.230000E+00 is 2.735290E+01
```

Sqrt -- Calculates the square root of a specified number.

FORMAT

sqrt(n) {**sqrt** is a function.}

Argument

n Any integer or real expression that evaluates to a number greater than zero.

Function Returns

The **sqrt** function returns a real value (even if n is an integer).

DESCRIPTION

The **sqrt** function calculates the square root of n.

EXAMPLE

```
PROGRAM sqrt_example;

VAR
   i_long   : integer32;
   r_single : single;
   r_double : double;

BEGIN
   write('Enter an integer -- '); readln(i_long);
   r_double := SQRT(i_long);
   writeln('The square root of ', i_long:1, ' is ', r_double);

   write('Enter a real number -- '); readln(r_single);
   r_double := SQRT(r_single);
   writeln('The square root of ', r_single, ' is ', r_double);
END.
```

Using This Example

Following is a sample run of the program named sqrt_example:

```
Enter an integer -- 24
The square root of 24 is    4.898979663848877E+00
Enter a real number -- 24.0
The square root of     2.400000E+01 is    4.898979663848877E+00
```

Statements

Throughout this encyclopedia, we refer to statements, both simple and compound. Here, we define statement.

Statement

When a format requires a statement, you must enter one of the following:

- An assignment statement like `x := 5` or `x := y + z`. An assignment statement can also be a call to a function like `x := ORD('a')`.
- A procedure call like `WRITELN('hi')`.
- A **goto**, **if/then**, **if/then/else**, **case**, **for**, **repeat**, **while**, **with**, **exit**, **next**, or **return** statement.
- A compound statement.
- An empty statement.

Simple and Compound Statements

When the format part of a listing in this chapter says that a command requires a simple statement, it just means that we require one statement (see above). A compound statement is a group of zero or more statements bracketed by the keywords **begin** and **end**. In other words, a compound statement has the following format:

```
begin
      statement1;
         .
         .
         .
      statementN
end;
```

The action part of a routine is itself a compound statement. A statement can be preceded by a label (but not every label is accessible; see the **goto** listing earlier in this chapter).

Succ -- Returns the successor of a specified ordinal value.

FORMAT

succ(x) {**succ** is a function.}

Argument

x	Must be an integer, Boolean, char, or enumerated expression.

Function Returns

The **succ** function returns a value having the same data type as x.

DESCRIPTION

The **succ** function returns the successor of x according to the following rules:

Data Type of x	**Succ Returns**
Integer	The numerical value equal to x + 1.
Boolean	True -- even if x already equals true.
Char	The character with the ASCII value one greater than the ASCII value of x.
Enumerated	The identifier to the right of x in the type declaration.

`SUCC(LASTOF(x))` generally is undefined; however, DOMAIN Pascal does not report an error. DOMAIN Pascal also doesn't report an error if you specify an integer value that is outside the range of the specified integer type. Therefore, your program should test for an out-of-bounds condition.

Compare the **succ** function to the **pred** function.

EXAMPLE

```
PROGRAM succ_example;

TYPE
   jours = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche);

VAR
   int  : integer;
   ch   : char;
   semaine : jours;

BEGIN
   int := succ(53);   writeln('The successor to 53 is ', int:1);
   ch := succ('q');   writeln('The successor to q is ', ch);
   semaine := succ(jeudi); writeln('The successor to jeudi is ', semaine:8);
END.
```

Using This Example

If you execute the sample program named succ_example, you get the following output:

```
The successor to 53 is 54
The successor to q is r
The successor to jeudi is VENDREDI
```

Then -- Refer to If earlier in this encyclopedia.

To -- Refer to For earlier in this encyclopedia.

Trunc -- Truncates a real number to an integer.

FORMAT

trunc(n) {**trunc** is a function.}

Argument

n	Any real value.

Function Returns

The **trunc** function returns an integer.

DESCRIPTION

The **trunc** function removes the fractional part of n to create an integer. (Compare **trunc** to **round**.)

EXAMPLE

```
PROGRAM trunc_example;
VAR
   x : REAL;
   y : INTEGER;

BEGIN
   x := 54.2;    y := TRUNC(x);    WRITELN(y);
   x := 54.5;    y := TRUNC(x);    WRITELN(y);
   x := 54.8;    y := TRUNC(x);    WRITELN(y);
END.
```

Using This Example

If you execute the sample program named `trunc_example`, you get the following output:

```
54
54
54
```

Compare these results to the results of executing program `round_example`.

Type Transfer Functions -- Permits you to change the data type of a variable or expression in the code portion of your program. (Extension)

FORMAT

transfer_function(x) {Type transfer functions are functions.}

Arguments

transfer_function	The name of any predeclared DOMAIN Pascal data type or any user-defined data type that has been declared in the program.
x	An expression.

DESCRIPTION

DOMAIN Pascal type transfer functions enable you to change the type of a variable or expression within a statement. To perform a type transfer function, use any user-created or standard type name as if it were a function name in order to "map" the value of its argument into that type.

With one exception, the size of the argument must be the same as the size of the destination type. (Chapter 3 describes the sizes of each data type.) This size equality is required because the type transfer function does not change any bits in the argument. DOMAIN Pascal just "sees" the argument as a value of the new type. The one exception is that integer subranges are always compatible regardless of their sizes.

It is important to remember that type transfer functions do not convert any value. Consider the following data type declarations:

```
VAR
    i : INTEGER32;
    r : REAL;
```

The following assignment *converts* the value of variable i to a floating-point number:

```
r := i;
```

However, in the following assignment, DOMAIN Pascal "sees" the bits in i as if they were actually representing a floating-point number. In this case, there is a transfer, but no conversion:

```
r := real(i);
```

Note that there are restrictions on the data types to which you can convert a given DOMAIN Pascal data type. For example, you get an error if you try the following:

```
i := r;
```

In such a case, there's no way for DOMAIN Pascal to know what you want to do with the portion of r after the decimal point. Use the **trunc** or **round** functions (described earlier in this encyclopedia) instead.

A practical application of type transfer functions is in controlling the bit precision of a computation. For example, consider the following program fragment:

```
VAR
   x, y : integer16;

BEGIN
   if x + y > 5
     then . . .
```

By default, the compiler expands operands x and y to 32-bit integers and performs 32-bit addition before making the comparison to 5. However, by using the following type transfer function, we can produce more efficient code:

```
VAR
   x, y : integer16;

BEGIN
   if INTEGER16(x + y) > 5
     then . . .
```

The disadvantage to using the type transfer function in the preceding fragment is that it ignores the possibility of integer overflow.

EXAMPLE

```
PROGRAM type_transfer_functions_example;
{ This program demonstrates two uses of type transfer functions. }

TYPE
   car_manufacturers = (volvo, fiat, nissan, dodge, porsche);
   pointer_to_word   = ^word;
   word              = array[1..10] of char;

VAR
   ordinal_value_of_car : integer16;
   car, actual_value_of_car : car_manufacturers;
   name, rename  : word  := [* of ' '];
   namepointer   : pointer_to_word;

BEGIN

{ Here, we convert an ordinal value into an enumerated value.}
       car := dodge;
       ordinal_value_of_car := ord(car);
       actual_value_of_car := CAR_MANUFACTURERS(ordinal_value_of_car);
       writeln('The actual value of ordinal value ', ordinal_value_of_car:1,
               ' is ', actual_value_of_car:7);

{ It is illegal to perform mathematical operations on a pointer variable.  }
{ However, by using type transfer functions you can temporarily make       }
{ a pointer variable into an integer32 variable so that you can perform    }
{ mathematical operations on it.  Then, after using the integer in a math  }
{ calculation, you can transfer the integer back to a pointer type by using}
{ a second type transfer function.  This routine prints the final eight    }
{ characters in the specified name.                                        }
       write('Enter a name that is 10 characters long -- '); readln(name);
       namepointer := addr(name);  {get starting address of name array}
       namepointer := UNIV_PTR(INTEGER32(namepointer) + 2);
       rename := namepointer^;
       writeln('The last eight characters of the name are ', rename:8);
END.
```

Using This Example

If you execute the sample program named ttf_example, you get the following output:

```
The actual value of ordinal value 3 is   DODGE
Enter a name that is 10 characters long -- CALIFORNIA
The last eight characters of the name are LIFORNIA
```

Unpack -- Copies a packed array to an unpacked array.

FORMAT

unpack(packed_array, unpacked_array, index) {**unpack** is a procedure.}

Arguments

unpacked_array	An array that has been defined without the keyword **packed**.
packed_array	An array that has been defined using the keyword **packed**.
index	A variable that is the same type as the array bounds (**integer**, **boolean**, **char**, or enumerated) of unpacked_array. Index designates the array element in unpacked_array to which **unpack** should begin copying.

DESCRIPTION

Unpack copies the elements in a packed array to an unpacked one. Data access with unpacked arrays generally is faster since data elements are always aligned on word boundaries.

Unpacked_array and packed_array must be of the same type, and for every element in packed_array, there must be an element in unpacked_array. That is, if you have the following **type** definitions

```
TYPE
    x : array[i..j] of single;
    y : packed array[m..n] of single;
```

the subscripts must meet these requirements:

j - index >= n - m {"index" as set in the call to **unpack**}

For example, it is legal to use **unpack** on two arrays defined like this:

```
TYPE
   big_array   : array[1..100] of integer;
   small_array : packed array[1..10] of integer;
VAR
   grande : big_array;
   petite : small_array;
```

You use index to indicate the array element in unpacked_array to which **unpack** should begin copying. For instance, given the previous variable declarations and assuming variable i is an **integer**, this fragment

```
i := 1;
unpack(petite, grande, i);
```

tells **unpack** to begin copying into grande[1]. **Unpack** keeps copying until it has exhausted all of petite's elements -- in this case 10. **Unpack** always copies all of its packed_array's elements, regardless of how many elements are defined for unpacked_array.

Index can take a value outside of packed_array's defined subscripts. That is, if in the example above, i equals 50, **unpack** copies these values this way:

```
grande[50]:= petite[1];
grande[51]:= petite[2];
          .
          .
          .
grande[59]:= petite[10];
```

See the listing for **pack** earlier in this encyclopedia.

EXAMPLE

```
PROGRAM unpack_example;

TYPE
    uarray = array[1..50] of integer16;
    parray = packed array[1..10] of integer16;

VAR
    full_range : uarray;
    sub_range  : parray;
    i, j       : integer16;

BEGIN
for i := 1 to 10 do
    sub_range[i] := i;
j := 30;

UNPACK(sub_range, full_range, j);
writeln ('The unpacked array now contains: ');
for i := 30 to 39 do
    writeln ('full_range[', i:2, '] = ', full_range[i]:2);

END.
```

Using This Example

If you execute the sample program named unpack_example, you get the following output:

```
The unpacked array now contains:
full_range[30] =  1
full_range[31] =  2
full_range[32] =  3
full_range[33] =  4
full_range[34] =  5
full_range[35] =  6
full_range[36] =  7
full_range[37] =  8
full_range[38] =  9
full_range[39] = 10
```

Until -- Refer to Repeat earlier in this encyclopedia.

While -- Execute the statements within a loop as long as the specified condition is true.

FORMAT

```
while condition do                      {while is a statement.}
        stmnt;
```

Arguments

condition	Any Boolean expression.
stmnt	A simple statement or a compound statement. (Refer to "Statements" earlier in this encyclopedia.)

DESCRIPTION

For, **repeat**, and **while** are the three looping statements of Pascal. With **while**, you specify the condition under which Pascal continues looping.

While marks the beginning of a loop. As long as condition evaluates to true, Pascal executes stmnt. When condition becomes false, Pascal transfers control to the first statement following the loop.

To jump out of a **while** loop prematurely (i.e., before the condition is true), do one of the following things:

- Use **exit** to transfer control to the first statement following the **while** loop.
- Use **goto** to transfer control outside the loop.

In addition to these measures, you can also call the **next** statement to skip the remainder of the statements in the loop for one iteration.

EXAMPLE

```
PROGRAM while_example;
{ This program contains two while loops. }
{ Compare it to repeat_example.          }
VAR
    num            : integer16;
    test_completed : boolean;
    i              : integer32;

BEGIN

    write('Enter an integer -- '); readln(num);
    WHILE (num < 101) DO
        BEGIN
        num := num + 10;
        writeln(num, sqr(num));
        END;

    writeln;
    test_completed := false;
    WHILE test_completed = false DO
        BEGIN
        write('Enter an integer (enter a 0 to stop the program) -- ');
        readln(i);
        if i = 0 then
            test_completed := true
        else
            writeln('The absolute value of ', i:1, ' is ', abs(i):1);
        END;
END.
```

Using This Example

Following is a sample run of the program named while_example:

```
Enter an integer -- 70
        80      6400
        90      8100
       100     10000
       110     12100

Enter an integer (enter a 0 to stop the program) -- 4
The absolute value of 4 is 4
Enter an integer (enter a 0 to stop the program) -- -5
The absolute value of -5 is 5
Enter an integer (enter a 0 to stop the program) -- 0

{Now, consider a second run of while_example. In contrast to repeat_example, }
{while_example does not execute the loop even once when x > 101.            }
Enter an integer -- 102

Enter an integer (enter a 0 to stop the program) -- 0
```

With -- Lets you abbreviate the name of a record.

FORMAT

With is a statement that takes the following format:

```
with v1, v2, ... vN do
       stmnt;
```

This format is equivalent to:

```
with v1 do
       with v2 do
              .
              .
              .
              with vN do
                     stmnt;
```

Arguments

v1	A record expression; that is, v1 must evaluate to a record. For example, it might be the name of a record, a pointer to a record, or a particular component in an array of records.
v2, ... vN	Optional record references or references that are qualified by v1, ... *v(N−1)*.
stmnt	A simple or compound statement. (Refer to the "Statements" listing earlier in this chapter.)

DESCRIPTION

Use **with** to abbreviate a reference to a field in a record. **With** works in the following manner. Suppose that X is a field within record MATHVALUES. Ordinarily, you must specify the full name MATHVALUES.X whenever you want to refer to the contents of field X. However, by using the statement

```
WITH MATHVALUES
```

you can simply specify X to refer to the contents of the field. Moreover, suppose that record MATHVALUES contains a field called TRIGONOMETRY which itself contains a field named Y. By specifying

```
WITH MATHVALUES, TRIGONOMETRY
```

you can refer to Y as Y rather than as MATHVALUES.TRIGONOMETRY.Y. Note that DOMAIN Pascal evaluates the expression v1 only once, and this evaluated expression is implied within the body of the **with** statement.

Now consider a fragment demonstrating **with**:

```
TYPE
   p = ^basketball_team;
VAR
   bb = basketball_team;
BEGIN
   WITH p^ DO
       BEGIN
            mascot     := 'tiger';
            p          := nil;
            height     := 198.2;
            bb.mascot  := 'lion';
       END;
.
.
.
```

Note two things about the preceding example. First, changing p does not affect access to the record identified by the **with** statement. Second, you can reference other records of the same type by completely qualifying the reference.

Extension to Standard Pascal

DOMAIN Pascal supports the standard format of **with** and also supports the following alternative format:

with v1:identifier1, *v2:identifier2, ... vN:identifierN* **do**
      stmnt;

This is very similar to the standard format for **with**. In this extension, the identifier is a pseudonym for the record reference v. To specify a record, use the identifier instead of the record reference v. Furthermore, to specify a field in a record, use identifier.field_name rather than merely field_name.

For example, given the following record declaration:

```
VAR
   basketball_team : record
      mascot     : array[1..15] of char;
      height     : single;
   end;
```

consider the following three methods of assigning values:

```
readln(basketball_team.mascot);       {Not using WITH.}
readln(basketball_team.height);       {Not using WITH.}

WITH basketball_team DO            {Using standard WITH.}
  begin
       readln(mascot);
       readln(height);
  end;

WITH basketball_team : B DO        {Using extended WITH.}
   begin
       readln(B.mascot);
       readln(B.height);
   end;
```

This feature is useful for working with long record names when two records contain fields that have the same names. (See the example at the end of this listing.)

EXAMPLE

```
PROGRAM with_example;
{ This program demonstrates the WITH statement. }

TYPE
    name = record
        first : array[1..10] of char;
        last : array[1..14] of char;
    end;
    documentation_department = record
        their_name        : name;
        current_project   : string;
    end;

VAR
    our_technical_writers, our_editors : documentation_department;

BEGIN
    writeln('In this routine, you enter data about Apollo documentors.');

{ First, we demonstrate the standard use of WITH. }
    WITH our_technical_writers, their_name DO
       BEGIN
          write('Enter the first name of the writer -- ');
          readln(first);

          write('Enter the last name of the writer -- ');
          readln(last);

          write('Enter a brief description of his or her current project-- ');
          readln(current_project);
       END; {with}

    writeln;

{ Next, we demonstrate the DOMAIN Pascal extensions to WITH. }
{ Use of the identifiers W and E permits a distinction between the records  }
{ inside the scope of the WITH statement. }
    WITH our_technical_writers : W, our_editors : E  DO
       BEGIN
          write('Enter the first name of the editor -- ');
          readln(E.their_name.first);

          write('Enter the last name of the editor -- ');
          readln(E.their_name.last);

          E.current_project := W.current_project;
       END;
END.
```

Using This Example

This program is available on-line and is named with_example.

Write, Writeln -- Writes the specified information to the specified file (or to the screen).

FORMAT

write(*f*, exp1:*field_width*, ..., *expN:field_width)*   {**write** and **writeln** are procedures.}

and

writeln(*f*, exp1:*field_width*, ..., *expN:field_width)*

Arguments

f — A variable having either the **text** or **file** data type. *F* is optional. If you do not specify *f*, DOMAIN Pascal writes to standard output (**output**) which is usually the transcript pad. (Note that **output** has a **text** data type.)

exp — One or more expressions separated by commas. An expression can be any of the following:

- A string constant
- An integer, real, char, Boolean, or enumerated expression
- An element of an array
- The name of an array variable whose base type is char
- A field of a record variable (assuming that the field is itself one of the previous four items)

Note that exp cannot be a set variable.

field_width — An integer expression that specifies the number of characters that **write** or **writeln** uses to output the value of this arg. *Field_width* is optional. Its effect depends on the data type of the exp to which it applies. (We detail these effects in the next section.) Note that you can specify *field_width* only if the *f* has the **text** data type.

DESCRIPTION

Write and **writeln** are output procedures. (**Put** is also an output procedure; see the **put** listing earlier in this encyclopedia.) **Write** and **writeln** both write the values of arguments exp1 through *expN* to the file specified by *f*. At runtime, Pascal writes the value of exp1 first, the value of *exp2* second, and so on until *expN*.

Write and **writeln** are identical in syntax and effect except that **writeln** appends a newline character after writing the exps but **write** does not. In addition, when using **write**, *f* can have a **file** or **text** data type; however, when using **writeln**, *f* must have a **text** data type only.

Before calling **write** or **writeln** to write to an external file, you must open the file for writing. Chapter 8 details this process. Note that you do not need to open the standard output (**output**) file before writing to it.

Following the call to **write**, *f^* is totally undefined.

The following paragraphs explain the output rules that DOMAIN Pascal uses to print the value of an exp.

Char Variables, Array of Char Variables, and String Constants

The following list shows the default *field_width*s for **char** variables, array of **char** variables, and string constants:

- If exp is a **char** variable, the default *field_width* is 1.
- If exp is an array of **char** variable, the default *field_width* is the declared length of the array. For example, if you declare an array named `Oslo_array` as

  ```
  Oslo_array : array[1..10] of char;
  ```

  then the default *field_width* is 10.
- If exp is a string constant, the default *field_width* is the number of characters in the string.

If you do specify a *field_width*, here's how **write** and **writeln** interpret it:

field_width	**What DOMAIN Pascal does**
= default	Writes a value with no leading or trailing blanks.
> default	Adds leading blanks.
< default	Truncates the excess characters at the end of the array of string.
= -1	Truncates any trailing blanks in the array. Standard Pascal issues an error if you specify a negative *field_width*.

For example, notice how the *field_width*s in the following **writeln** statements affect output. (The first two lines of output form a column ruler to help you notice columns.)

DOMAIN Pascal Statements *Output*

```
VAR
name     : array[1..20] of char;
          grade    : char;

BEGIN
   name := 'Zonker Harris        ';
   grade := 'F';
                                                  1         2         3
                                         123456789012345678901234567890
   WRITELN(name, grade);                 Zonker Harris       F
   WRITELN(name:-1, grade);              Zonker HarrisF
   WRITELN(name:4, grade);               ZonkF
   WRITELN(name:25, grade);                   Zonker Harris       F
```

Integer Values

The default *field_width* for an integer value is 10 spaces. This default applies to **integer**, **integer16**, **integer32**, and subrange variables, and to elements of an array that have one of these types as a base type. It also applies to record fields that have one of the aforementioned types.

If you specify a *field_width* greater than the number of digits in the integer value, DOMAIN Pascal prints the value with leading blanks.

If you specify a *field_width* less than or equal to the number of digits in the integer value, DOMAIN Pascal writes the value without leading or trailing blanks. Note that specifying a *field_width* never causes DOMAIN Pascal to truncate the written value.

For example, consider an **integer16** variable named `small_int` with a value of 452 and an **integer32** variable named `large_int` with a value of 70,600,100. Notice how the *field_widths* in the following **writeln** statements affect output. (The first two lines of output form a column ruler to help you notice columns.)

DOMAIN Pascal Statements *Output*

```
                                          1         2         3
                                 123456789012345678901234567890
WRITELN(small_int);                     452
WRITELN(large_int);                70600100
WRITELN(small_int:5);                 452
WRITELN(small_int:1);            452
```

Real Values

For real exp you can supply no *field_width*, a one-part *field_width*, or a two-part *field_width*. Here are the rules:

- If you don't supply a *field_width*, DOMAIN Pascal uses 13 spaces to write a single-precision value and 22 spaces to write a double-precision value.
- If you supply a one-part *field_width*, DOMAIN Pascal adds or removes digits from the fractional part.
- If you supply a two-part *field_width*, DOMAIN Pascal interprets the first part of the *field_width* as the total number of characters to print and the second part of the *field_width* as

the number of digits to print following the decimal point. Note that the second part of the *field_width* has priority over the first part. For instance, suppose that you request a total width (the first part) of 5 characters and a fractional width (the second part) of 7 characters. Since DOMAIN Pascal cannot satisfy both parts, it will satisfy only the second part.

If you don't supply a two-part *field_width*, DOMAIN Pascal always leaves one leading space for positive numbers; none for negative numbers.

If there is not enough room for all the digits in the number, DOMAIN Pascal rounds the value rather than truncating it.

For example, suppose that a single-precision real variable named `velocity` has a value of 43.54893. The following table shows how various **writeln** statements affect output. (The first two lines of output form a column ruler to help you notice columns.)

```
DOMAIN Pascal Statements             Output

                                            1         2         3
                                   123456789012345678901234567890

WRITELN(velocity);                  4.354893E+01
WRITELN(velocity:20);               4.3548930000000E+01
WRITELN(velocity:1);                4.4E+01
WRITELN(velocity:15:4);                    43.5489
WRITELN(velocity:7:4);             43.5489
WRITELN(velocity:7:2);               43.55
WRITELN(velocity:3:0);             44.
WRITELN(velocity:1:5);             43.54893
```

Enumerated and Boolean Values

DOMAIN Pascal keeps the same rules for writing enumerated and Boolean values. For both types, the default *field_width* is 15. Here's what happens if you specify your own *field_width*:

- If you specify a *field_width* less than 15, DOMAIN Pascal subtracts a suitable number of leading blanks.
- If you specify a *field_width* greater than 15, DOMAIN Pascal adds a suitable number of leading blanks.

Note that DOMAIN Pascal never truncates any of the characters in the value (even if the *field_width* is less than the number of characters).

The following example shows how various *field_widths* affect output. (The first two lines of output form a column ruler to help you notice columns.)

DOMAIN Pascal Statements   *Output*

```
VAR
   colors : (red, brown, magenta);
   evil   : boolean;
                                              1         2         3
                                     123456789012345678901234567890

BEGIN
   colors := brown;
   WRITELN(colors);                            brown
   WRITELN(colors:8);                   brown
   WRITELN(colors:1);                brown

   evil := true;
   WRITELN(evil);                               true
   WRITELN(evil:2);                  true
```

EXAMPLE

```
PROGRAM write_example;

{ This example reads one input line from the keyboard and writes it to }
{ filename 'truth'.}

CONST
   pathname = 'truth';

VAR
   a_line   : string;
   wisdom   : text;
   statint  : integer32;

BEGIN

   open(wisdom, pathname, 'NEW', statint);
   if statint <> 0 then
      return
      else rewrite(wisdom);

   WRITE('Enter a sentence of truth -- ');
   readln(a_line);
   WRITELN(wisdom, a_line);
   WRITELN(wisdom, chr(10), chr(10), chr(9), 'Thank You', chr(7));
{ ASCII 10 is a linefeed.  ASCII 9 is a tab.  ASCII 7 is the bell. }

END.
```

Using This Example

This program is available on-line and is named write_example.

Xor -- Returns the exclusive or of two integers. (Extension)

FORMAT

xor(int1, int2) {**xor** is a function.}

Arguments

int1, int2	Integer expressions.

Function Returns

The **xor** function returns an integer value.

DESCRIPTION

Use the **xor** function to take the bitwise exclusive or of int1 and int2. The **xor** function belongs to the bitwise class consisting of &, !, and ~, and *does not* belong to the Boolean operator class consisting of **and**, **or**, and **not**. When matching bits for an **xor** function, DOMAIN Pascal uses the following truth table:

bit x of op1	**bit x of op2**	**bit x of result**
0	0	0
0	1	1
1	0	1
1	1	0

EXAMPLE

```
PROGRAM xor_example;

{This program finds the exclusive or of two integers by using XOR. }

VAR
   i1, i2, result : integer16;

BEGIN
   write('Enter an integer -- '); readln(i1);
   write('Enter another integer -- '); readln(i2);
   result := XOR(i1, i2);
   writeln('The exclusive or of ', i1:1, ' and ', i2:1, ' is ',
            result:1);
END.
```

Using This Example

Following is a sample run of the program named xor_example:

```
Enter an integer -- 6
Enter another integer -- 20
The exclusive or of 6 and 20 is 18
```

Chapter 5

Procedures and Functions

This chapter explains how to declare and call procedures and functions. The term "routine", which appears throughout this chapter, means either procedure or function. The terms "parameter" and "argument" also appear throughout this chapter. In this case, argument means the data passed to a routine, while parameter means the templates for the data that the called routine receives.

Chapter 2 mentioned that routine headings take the following format:

attribute_list **procedure** name (*parameter_list); routine_options*;

or

attribute_list **function** name (*parameter_list*) : data_type; *routine_options*

This chapter details the *parameter_list, routine_options*, and *attribute_list.*

5.1 Parameter List

You can declare a routine with or without parameters. If you declare it without parameters, you cannot pass any arguments to the routine. If you declare it with parameters, you specify the data type of each argument that can be passed to the routine.

You specify parameters within a parameter list. You can specify a maximum of 65 parameters within the list. A parameter list has the following format:

```
(param_type1  par_list1 : data_type1;
      .           .          .
      .           .          .
      .           .          .
  param_typeN  par_listN : data_typeN);
```

The *param_type* is optional; for information, see the "Parameter Types" section later in this chapter.

A par_list consists of one or more parameters that have the same data_type. Thus, the combination

```
par_list : data_type
```

is similar to a variable declaration. Like a variable declaration, each parameter in par_list must be a valid identifier. Also, each data type must be a predeclared DOMAIN Pascal or user-defined data type. That is, you cannot specify an anonymous data type. (See the "Var Declaration Part" section in Chapter 2 for more information on anonymous data types.)

You can use a parameter in the action part of the routine just as you would use any variable. Consider the following sample routine declarations:

```
{ Declare a procedure with no parameter list. }
    Procedure simple;


{ Declare a procedure with a parameter list that has two parameters. }
    Procedure con(a : integer;
                  b : real);


{ Declare a function with a parameter list that has two parameters }
{ sharing the same data type. }
    Function anger(x,y : boolean) : integer16;


{ Declare a function with a parameter list that has three parameters. }
    Function big(quart  : integer16;
                 volume : real;
                 cost   : single) : single;
```

The following routine declaration is wrong because it uses an anonymous data type:

```
    Procedure range(small_range : 0..10);              {WRONG!}
```

To call a routine, simply specify its name. If the procedure has a parameter list, you must also specify arguments. The data type of each argument, with two exceptions, must match the data type of its corresponding parameter. For example, if the second parameter is declared as **integer16**, the second argument must be an **integer16** value. The following examples call the routines that were previously declared:

```
{ Call simple with no arguments. }
    simple;


{ Call con with two arguments.  The first argument must be an integer, }
{ and the second argument must be a real number. }
    con(14, 5.2);


{ Call anger with two Boolean arguments.  Variable answer must have    }
{ been declared as an integer.                                         }
    answer := anger(true, false);


{ Call big with three arguments.  Assume that pints is an integer and  }
{ price is a real number.                                              }
   if big(pints * 2, 4.23E3, price) > 1.40
      then ...
```

DOMAIN Pascal supports many features that let you specify precisely how a routine is to be called. The remainder of this chapter details these features.

5.2 Parameter Types

A *param_type* is optional. If you do not include one, you are in effect passing a value parameter. Value parameters are discussed in this section. If you want to specify a *param_type*, it must be one of the following:

- **var** (i.e., a variable parameter)
- **in**
- **out**
- **in out**

The following subsections describe each of these.

5.2.1 Variable Parameters and Value Parameters

In standard Pascal, you pass arguments to and from routines as variable parameters or value parameters. (DOMAIN Pascal supports both methods plus certain extensions described later in this subsection.) The following examples illustrate the distinction between variable parameters and value parameters.

Pascal regards variable parameters as synonyms for the variable you pass to them. In var_parameter_example below, variable parameter n becomes a synonym for argument x; that is, whatever happens to n in addc also happens to x. Note that you can only pass a variable as an argument to a variable parameter. You cannot pass a value.

Pascal does not regard value parameters as synonyms to the arguments you pass to them. In value_parameter_example, value parameter n takes on a copy of the value of x within addc; therefore, whatever happens to n in addc has no affect on the value of x. Note that you can pass variables, values, or expressions as arguments to a routine with value parameters.

```
Program var_parameter_example;

VAR
   x : integer16;


PROCEDURE addc(VAR n : integer16);
BEGIN
   n := n + 100;
   writeln('In addc, n=',n:4);
END;


BEGIN {main}
   x := +10;
   addc(x);
   writeln('In main, x=',x:4);
END.
```

```
Program value_parameter_example;

VAR
   x : integer16;


PROCEDURE addc(n : integer16);
BEGIN
   n := n + 100;
   writeln('In addc, n=',n:4);
END;


BEGIN {main}
   x := +10;
   addc(x);
   writeln('In main, x=',x:4);
END.
```

Both programs are available on-line and are named var_parameter_example and value_parameter_example.

Here are the results you get from each program:

Execution of var_parameter_example

```
In addc, n= 110
In main, x= 110
```

Execution of value_parameter_example

```
In addc, n= 110
In main, x=  10
```

The only difference between the two programs is the keyword **var** in the procedure declaration statement of var_parameter_example. This keyword identifies n as a variable parameter; the absence of **var** identifies n as a value parameter.

5.2.2 In, Out, and In Out -- Extension

In standard Pascal, you cannot specify the direction of parameter passing. However, DOMAIN Pascal supports extensions to overcome this problem. You can use the following keywords in your routine declaration:

- **In** -- This keyword tells the compiler that you are going to pass a value to this parameter, and that the routine is not allowed to alter its value. If the called routine does attempt to change its value (that is, use the parameter on the left side of an assignment statement), the compiler issues an "Assignment to IN argument" error.

- **Out** -- This keyword tells the compiler that you are not going to pass a value to the parameter, but that you expect the routine to assign a value to the parameter. It is incorrect to try to use the parameter before the routine has assigned a value to it,, although the compiler does not issue a warning or error in this case.

  If the called routine does not attempt to assign a value to the parameter, the compiler may issue a "Variable was not initialized before this use" warning. This could occur if your routine only assigns a value to the parameter under certain conditions. If that is the case, you should designate the parameter as **var** instead of **out.**

  In some cases, the compiler cannot determine whether or not all paths leading to an **out** parameter assign a value to it. If that happens, the compiler does not issue a warning message.

- **In out** -- This keyword tells the compiler that you are going to pass a value to the parameter, and that the called routine is permitted to modify this value. It is incorrect to call the routine before assigning a value to the parameter, although the compiler does not issue a warning or error in this case. The compiler also doesn't complain if the called routine does not attempt to modify this value.

For example, consider the following program:

```
PROGRAM in_out_example;
{This program demonstrates the IN, OUT, and IN OUT parameters.}
VAR
    leg1, leg2     : integer16;
    hypotenuse : single;
    temp  : real;
    unit  : char;


    PROCEDURE pythagoras(IN         leg1 : integer16;
                         IN         leg2 : integer16;
                         OUT  hypotenuse : single);
    BEGIN
       hypotenuse := sqrt((leg1 * leg1) + (leg2 * leg2));
    END;
```

```
      FUNCTION boiling(IN OUT  temp : real;
                       IN      unit : char) : boolean;
      BEGIN
         if unit = 'F'
            then temp := (temp - 32) * 0.55555;
         if temp >= 100
            then boiling := true
         else boiling := false;
      END;


  BEGIN
     write('Enter the first leg of a triangle -- ');   readln(leg1);
     write('Enter the other leg of the triangle -- '); readln(leg2);
     pythagoras(leg1, leg2, hypotenuse);
     writeln('Hypotenuse of the triange is ', hypotenuse);

     writeln(chr(10), chr(10), 'Assume 1 Atm. pressure');
     write('Enter the water temperature -- '); readln(temp);
     write('Is this temp. in Fahrenheit or Celsius (F or C) -- ');
     readln(unit);

     if boiling(temp, unit)
        then writeln(temp:5:1, ' degree water will boil!')
        else writeln(temp:5:1, ' degree water will not boil.');

  END.
```

This program is available on-line and is named in_out_example.

NOTE: The compiler checks for misuses of **in**, **out**, and **in out** at compiletime, but the system does not check for such errors at runtime.

5.2.3 Univ -- Universal Parameter Specification -- Extension

Univ is a special parameter type that you specify immediately prior to the data_type (rather than prior to the par_list).

By default, DOMAIN Pascal checks that the argument you pass to a routine has the same data type as the parameter you defined for the routine. However, you can tell DOMAIN Pascal to suppress this type checking. You do this by using the keyword **univ** prior to a type name in a parameter list. By using **univ**, you can pass an argument that has a different data type than its corresponding parameter.

Univ is especially useful for passing arrays. For example, the following program would be incorrect without the keyword **univ**. That's because `little_array` and `big_array` have different data types:

```
TYPE
    big_array     = array[1..50] of integer32;

VAR
    large_array  : array[1..50] of integer32;
    medium_array : array[1..25] of integer32;
    little_array : array[1..10] of integer32;
    sum          : integer32;

    Procedure sum_elements(in          b : UNIV big_array;
                           in array_size :      integer16;
                           out       sum :      integer32);
    BEGIN
        . . .
    END;

BEGIN  {main}
   sum_elements(little_array, 10, sum);
END.
```

In addition to the procedure call listed above, you could also make either of the following calls to procedure `sum_elements`:

```
sum_elements(medium_array, 25, sum);
```

or

```
sum_elements(large_array, 50, sum);
```

Use **univ** carefully! It can cause problems if improperly used. The most frequent source of trouble is a difference in size between the argument and parameter data types. The data type of the parameter determines how the called routine treats the data passed to it. Typically, routines that use **univ** parameters have another parameter that supplies additional information about the size or type of the actual parameter. In the preceding example, the `array_size` parameter gives the size of the array parameter passed. The following example shows another possible misuse of **univ**:

```
Program univ_example;

{This example demonstrates poor use of UNIV.}
{The program uses UNIV to pass two double-precision arguments to two}
{single-precision parameters.  The calculation of 'mean' will not be}
{correct because single- and double-precision real numbers have     }
{different bit patterns for exponent and mantissa.  Furthermore,    }
{the compiler will not warn you about this problem.                 }

VAR
   first_value, second_value : double;


     Procedure average(s,t : UNIV single);
     VAR
        mean : double;
     BEGIN
        mean := (s + t) / 2.0;
        writeln('The average is ', mean);
     END;


BEGIN {main}
    write('Enter the first  value -- '); readln(first_value);
    write('Enter the second value -- '); readln(second_value);
    average(first_value, second_value);
END.
```

NOTE: To prevent some problems that result from suppressing type checking, explicitly declare **univ** parameters as **in, out, in out,** or **var.**

When you pass an expression argument (as opposed to a variable argument) to a **univ** parameter, DOMAIN Pascal extends the expression to be the same size as the **univ** parameter. In addition, the compiler issues the following message:

Expression passed to UNIV formal *NAME* was converted to *NEWTYPE.*

5.3 Routine Options

As mentioned in the beginning of this chapter, you can optionally specify *routine_options* at the end of the routine declaration. DOMAIN Pascal supports the following routine options:

- **forward**
- **extern**
- **internal**
- **variable**
- **abnormal**
- **val_param**
- **nosave**
- **noreturn**
- **d0_return**

Use one of the following formats to specify one or more of these options:

routine_option1; ... routine_optionN;

or

options(*routine_option1, . . . routine_optionN);*

The two formats are equivalent; the first format is shorter, and the second format is more readable. Note, however, that you can only use **format**, **extern**, **internal**, and **val_param** with the first format.

Here are some examples:

```
FUNCTION  eggs_and_ham(letter : char) : char;  INTERNAL;
```

or

```
FUNCTION  eggs_and_ham(letter : char) : char;  OPTIONS(INTERNAL);

PROCEDURE sam_i_am(x, y : real);   EXTERN; ABNORMAL;
```

or

```
PROCEDURE sam_i_am(x, y : real);   OPTIONS(EXTERN, ABNORMAL);
```

The remainder of this section explains the routine options.

5.3.1 Forward

The **forward** option is a feature of standard Pascal and DOMAIN Pascal. By default, you can only call a routine that was previously declared in the program. The **forward** option tells the compiler that the procedure is declared past (forward) of the statement that calls it.

For example, in the following program, procedure `convert_degrees_to_radians` is declared as **forward**. This allows procedure `find_tangent` to call procedure `convert_degrees_to_radians` even though `find_tangent` precedes it in the file.

```
PROGRAM forward_example;

{This program demonstrates the FORWARD attribute.}

Function convert_degrees_to_radians(d : real) : real; FORWARD;

VAR
   degrees, tangent : real;


Procedure find_tangent(IN  degrees : real;
                       out tangent : real);
VAR
   radians : real;
BEGIN
   radians := convert_degrees_to_radians(degrees);
   tangent :=  sin(radians)  /  cos(radians);
END;


Function convert_degrees_to_radians; { No parameters here! }
CONST
    degrees_per_radian = 57.2958;
BEGIN
    convert_degrees_to_radians := (d / degrees_per_radian);
END;


BEGIN
   write('Enter a value in degrees -- ');
   readln(degrees);
   find_tangent(degrees, tangent);
   writeln('The tangent of ', degrees:6:3, ' is ', tangent:6:3);
END.
```

This program is available on-line and is named `forward_example`.

If it weren't for the **forward** option, the compiler would issue the following error:

```
CONVERT_DEGREES_TO_RADIANS has not been declared in routine
FIND_TANGENT
```

Note that this program declares procedure `convert_degrees_to_radians`, its parameters, and the **forward** option in the declaration part of the main program, not in the routine heading. The routine heading declares procedure `convert_degrees_to_radians` without declaring parameters or options. You must declare routines this way when using the **forward** option.

5.3.2 Extern -- Extension

Extern is an extension to standard Pascal. It tells the compiler that the routine is possibly defined outside of this source code file. (Chapter 7 details **extern**. See also **define**, which is also detailed in Chapter 7.)

5.3.3 Internal -- Extension

Internal is an extension to standard Pascal. By default, all top-level routines defined in a module become global symbols. But if you declare the routine with the **internal** option, the compiler makes the routine a local symbol. (Chapter 7 details **internal.**)

5.3.4 Variable -- Extension

Variable is an extension to standard Pascal. By default, you must pass the same number of arguments to a routine each time you call the routine. However, the **variable** option allows you to pass a variable number of arguments to the routine. You might want to specify an argument count as the first parameter.

For example, consider the following program.

```
PROGRAM variable_attribute_example;

{ This program demonstrates the routine attribute called VARIABLE which }
{ allows you to pass a variable number of arguments to a routine.       }

VAR
   first_value, second_value : real;
   precision : real;
   answer   : char;


Procedure average(arg_count : integer16;
                  d1, d2    : real;
                  p : real);
                  options(VARIABLE); {We can pass up to four arguments.}
VAR
   mean : real;
BEGIN
   mean := (d1 + d2) / 2.0;
   if arg_count = 3
       then writeln('The mean is ', mean:4:1, ' to a precision of 0.1')
   else if (arg_count = 4) and (p = 0.01)
       then writeln('The mean is ', mean:4:2, ' to a precision of .01')
   else if (arg_count = 4) and (p = 0.001)
       then writeln('The mean is ', mean:4:3, ' to a precision of .001')
   else
       writeln('Improper argument count or precision');
END;



BEGIN {main}
    writeln('This program calculates the mean of two real numbers.');
    write('Enter the first  value -- '); readln(first_value);
    write('Enter the second value -- '); readln(second_value);
    write('Do you want to specify a precision (enter y or n) -- ');
    readln(answer);
    if answer = 'y' then
        begin
            write('Please enter the precision (.01 or .001) -- ');
            readln(precision);
            average(4, first_value, second_value, precision);
        end
    else    average(3, first_value, second_value);
END.
```

This program is available on-line and is named `variable_attribute_example`.

5.3.5 Abnormal -- Extension

Abnormal is an extension to standard Pascal. It warns the compiler that a routine can cause an abnormal transfer of control. This option affects the way the compiler optimizes the calling routine, but does not affect the way the compiler optimizes the called routine (i.e., the routine that is declared **abnormal**).

For example, the following use of **abnormal** causes the compiler to be careful about optimizing around any cleanup handler:

```
FUNCTION pfm_$cleanup(clean_up_record) : status_$T; OPTIONS(ABNORMAL);
```

5.3.6 Val_param -- Extension

Val_param is an extension to standard Pascal. By default, Pascal passes arguments by reference. However, when you use the **val_param** option, you tell DOMAIN Pascal to pass arguments by value when possible.

This option is useful when you are writing a routine that calls a DOMAIN C routine, since C passes all arguments by value. See Chapter 7 and the *DOMAIN C Language Reference* for more details.

5.3.7 Nosave -- Extension

Nosave is an extension to standard Pascal. You should use it with a Pascal program call to an assembly language routine that doesn't follow the usual conventions for preserving these registers:

- Data registers D2 through D7
- Address registers A2 through A4
- Floating-point registers FP2 through FP7

Nosave indicates that the contents of these registers will not be saved when the assembly language routine finishes executing and returns to the Pascal program. However, the assembly language routine must *always* preserve register A5, which holds the pointer to the current stack area. It also must *always* preserve A6, which holds the address of the current stack frame. That is, the called routine must preserve A5 and A6 even if you use **nosave**.

5.3.8 Noreturn -- Extension

Noreturn is an extension to standard Pascal. This routine specifies an unconditional transfer of control; once a procedure or function with **noreturn** is called, control can never return to the caller. The routine marked **noreturn** is executed, and the program terminates.

When you specify this keyword, the compiler may optimize the code it generates so that any return sequence or stack adjustments after the call to the routine marked **noreturn** are eliminated as being unreachable code.

5.3.9 D0_return -- Extension

D0_return is an extension to standard Pascal. By default, a Pascal function returning the value of a pointer type variable puts that value in address register A0. **D0_return** causes the compiler to put the value in A0 *and* data register D0.

You should use **d0_return** if your Pascal function returns a pointer to a C or FORTRAN program, because those languages expect to see function results in D0. The option has no effect on any Pascal routines that call the function. Those Pascal routines should still expect to see the function's result returned in A0.

Note: The second character in this option is a zero, not a capital O.

5.4 Attribute List -- Extension

As noted in the beginning of this chapter, you can declare an optional routine *attribute_list* at the beginning of a routine heading. With this list, you can specify a nondefault section name for the code and data of a routine, or you can prevent the compiler from incorrectly optimizing nested **internal** procedures. The *attribute_list* affects a routine body, while the *routine_options* affect the routine interface.

The *attribute_list* consists of one or more routine attributes enclosed by brackets. DOMAIN Pascal currently supports the **section** routine attribute.

5.4.1 Section -- Extension

By using the routine attribute **section**, you can specify a nondefault section name for the code and data in a routine. A "section" is a named contiguous area of an executing object. (Refer to the *DOMAIN Binder and Librarian Reference* for full details on sections.) By default, the compiler assigns code to the PROCEDURE$ section and data to the DATA$ section. Thus, by default, all code from every routine in the program is assigned to PROCEDURE$, and all static data from every routine in the program is assigned to DATA$. However, DOMAIN Pascal permits you to override the default of PROCEDURE$ and DATA$ on a routine-by-routine basis. (You can also override the defaults on a variable-by-variable or module-by-module basis.) This makes it possible to organize the runtime placement of routines so that logically related routines can share the same page of main memory and thus reduce page faults. Likewise, you can declare a rarely-called routine as being in a separate section from the frequently-called routines.

To override the default sections, preface your routine heading with a phrase of the following format:

```
[section( codesect, datasect )] procedure . . .

or

[section( codesect, datasect )] function . . .
```

If you omit either the codesect or the datasect, the present default continues to take effect. For example, consider the following fragment:

```
Program example;
   VAR stat: integer;          { In DATA$ }

PROCEDURE top;                 { In PROCEDURE$}
   VAR datl : static integer;  { IN DATA$ }
   BEGIN
        ...                    { In PROCEDURE$}
   END;
[SECTION(npc,npd)] FUNCTION foobarl : REAL;   { In "npc" }
   VAR seed : static real;                    { In "npd" }
   BEGIN
        ...                                   { In "npc" }
   END;

[SECTION(error_rout_c, error_rout_d)] PROCEDURE xxx; { In error_rout_c }
   VAR check : static integer32;                     { In error_rout_d }
   BEGIN
        ...                                   { In error_rout_c }
   END;


FUNCTION regular : integer; { In PROCEDURE$ }
   VAR dat2 : static real;  { In DATA$      }
   BEGIN
        ...                 { In PROCEDURE$ }
   END;

[SECTION(npc,npd)] PROCEDURE foobar2;         { In "npc" }
   VAR seedling : static real;                { In "npd" }
   BEGIN
        ...                                   { In "npc" }
   END;


BEGIN {main}
        ...                  { In PROCEDURE$ }
END;
```

Nested routines inherit the section definitions of their outer routine unless they specify their own section definitions. For example, if the foobarl function contained nested routines, DOMAIN Pascal defaults to placing their code and static data into the "npc" and "npd" sections respectively.

5.5 Recursion

A recursive routine is a routine that calls itself. DOMAIN Pascal, like standard Pascal, supports recursive routines. The following example demonstrates a recursive method for calculating factorials:

```
PROGRAM recursive_example;
{ Demonstrates recursion by calculating a factorial. }

VAR
   x, y : integer32;

Function factorial(n : integer32) : integer32;
BEGIN
   if n = 0
     then factorial := 1
     else factorial := n * factorial(n-1);   {factorial calls itself.}
END;


BEGIN {main}
   writeln('This program finds the factorial of a specified integer.');
   write('Enter a positive integer (from 0 to 16) -- ');  readln(x);
   y := factorial(x);
   writeln('The factorial of ', x:1, ' is ', y:1);
END.
```

Chapter 6

Program Development

This chapter describes how to produce an executable object file (i.e., finished program) from DOMAIN Pascal source code. Briefly, you create an executable object file in the following steps:

1. Compile all source code files that make up the program. The compiler creates one object file for each source code file.

2. If your program consists of multiple object files, you must bind them together with the DOMAIN binder utility. If your program consists of only one object file, there is no need to bind. The DOMAIN binder resolves external references; that is, it connects the different object files so that they can communicate with one another. Before binding, you may wish to package related object files into a library file with the DOMAIN librarian utility.

3. You can either debug your program (with the DOMAIN language level debugger) or you can execute it.

Figure 6-1 illustrates this process.

This chapter details the compiler and provides brief overviews of the binder, librarian, and debugger.

In addition to this process, you can also use the DOMAIN Software Engineering Environment (DSEE) system to develop DOMAIN Pascal programs. This chapter provides a short overview of the DSEE system. Also, this chapter contains an introduction to DOMAIN/Dialogue, which is a product that simplifies the writing of user interfaces.

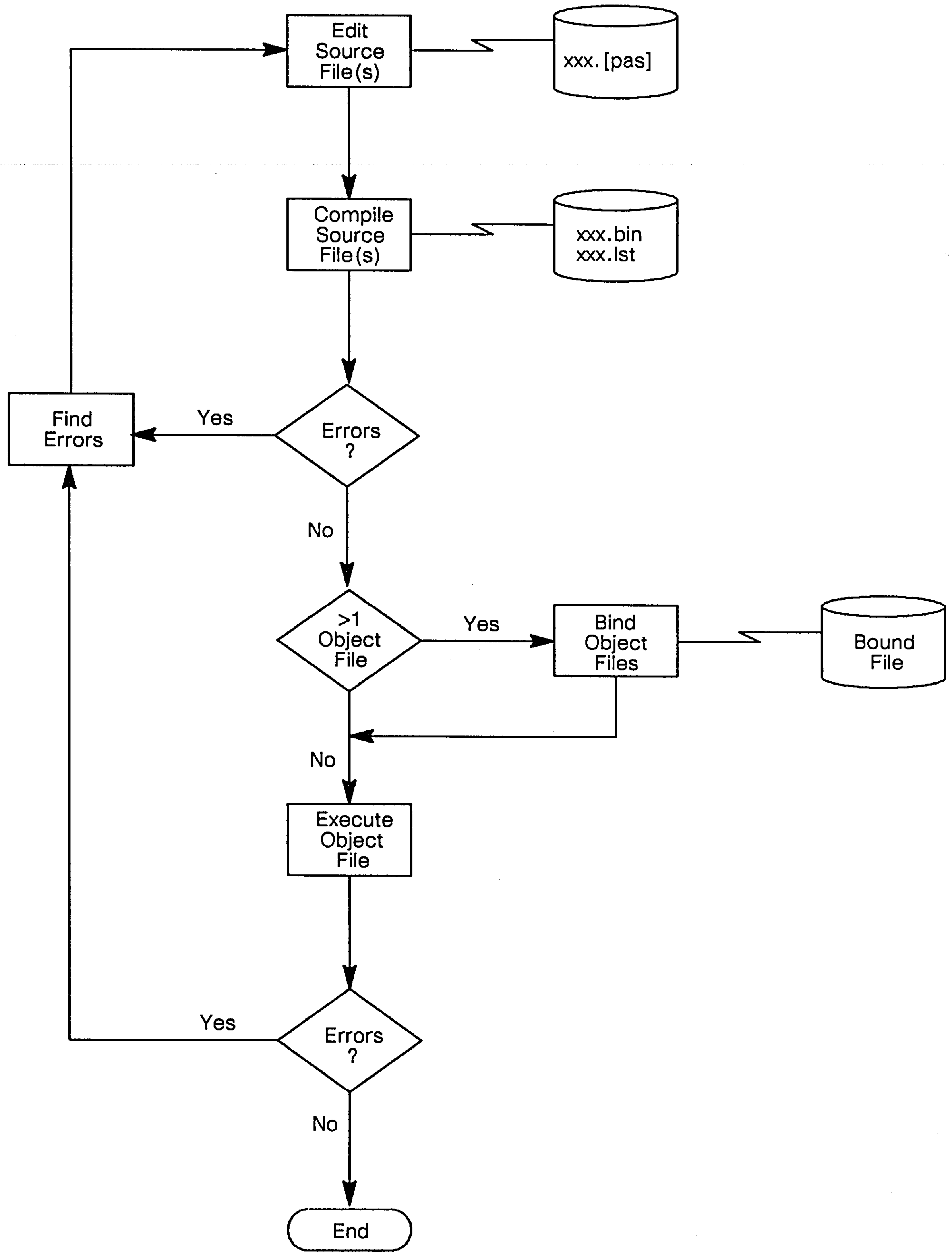

Figure 6-1. Steps in DOMAIN Pascal Program Development

6.1 Compiling

You compile a file of DOMAIN Pascal source code by entering a command of the following format in any AEGIS or DOMAIN/IX Shell:

```
$ pas source_pathname  option1 ... optionN
```

(Throughout this chapter we show commands with a variety of shell prompts. All commands work exactly the same in any shell, although, of course, you must remember that DOMAIN/IX shells are case-sensitive.)

Source_pathname is the pathname of the source file you want to compile. You can compile only one source file at a time. In order to simplify your search for Pascal source programs, we recommend that source_pathname end with a **.pas** suffix, but if you use the suffix, you need not specify it in the compile command line.

Your compile command line can contain one or more of the *options* listed in Table 6-1. Note that you cannot abbreviate these options.

For example, consider the following three sample compile command lines, all of which compile source code file circles.pas:

```
no options   -->   $ pas circles
one option   -->   % pas circles -l
four options -->   $ PAS circles -map -exp -cond -cpu 460
```

6.1.1 Compiler Output

If there are no errors in the source code and the compilation proceeds normally, the compiler creates an object file in your current working directory.

By default, DOMAIN Pascal gives the object file the same pathname as the source pathname, but with the **.bin** suffix. For example, if your source code is stored in file test.first, the following command line produces an object file called test.first.bin in your working directory.

```
$ pas test.first
```

There is one exception. If your source_pathname ends with **.pas**, the compiler *replaces* that suffix with **.bin**. So if your source code is stored in extratest.pas, the following command produces the object file extratest.bin in your working directory:

```
$ pas extratest
```

If you want the object file to have a nondefault name, use the **-b** *pathname* option. For example:

```
$ PAS test  -b //good/compilers/newtest
```

The preceding command produces an object file called newtest.bin in directory //good/compilers.

6.2 Compiler Options

DOMAIN Pascal supports a variety of compiler options. Table 6-1 summarizes the options, while the following sections describe all the options in detail.

Table 6-1. DOMAIN Pascal Compiler Options

Option	What It Causes the Compiler to Do
☆ **-align**	Align data on longword boundaries. This helps the DNx60 processors execute programs more efficiently.
-nalign	Suppress alignment.
☆ **-b** *pathname*	Generate a binary file (that is, an executable file) at program_name.bin or *pathname.bin.*
-nb	Suppress creation of binary file.
-comchk	Issue a warning if comments are not paired correctly.
☆ **-ncomchk**	Suppress checking for paired comments.
-cond	Compile lines prefixed with the **%debug** compiler directive.
☆ **-ncond**	Ignore lines prefixed with the **%debug** compiler directive.
-config var1... *varN*	Set special conditional compilation variables to true.
-cpu id	Generate code for a particular CPU. Values for id are: 160, 460, 560, 660, 330, 90, 570, 580, 3000, any and peb. Any is the default.
-ndb	Suppress creating debugging information. The debugger cannot debug such a program.
☆ **-db**	Generate minimal debugging information. When you debug this program, you can set breakpoints, but you can't examine variables.
-dbs	Generate full runtime debug information and optimize the source code in the executable object file. (Implies **-opt 3.**)
-dba	Generate full runtime debug information but don't optimize the source code in the executable object file.
-exp	Generate assembly language listing (implies -l).
☆ **-nexp**	Suppress creating assembly language listing.
-idir dir1... *dirN*	Search for an include file in alternate directories.
-iso	Compile the program using ISO/ANSI Standard Pascal rules for certain DOMAIN Pascal features that deviate from the standard.
☆ **-niso**	Compile the program using DOMAIN Pascal features.
-l *pathname*	Generate a listing file at program_name.lst or *pathname.lst.*
☆ **-nl**	Suppress creation of listing file.

☆ denotes a default option.

Table 6-1. DOMAIN Pascal Compiler Options (continued)

Option	What It Causes the Compiler to Do
-map	Generate symbol table map (implies -l).
☆ -nmap	Suppress creation of symbol table map.
☆ -msgs	Generate final error and warning count message.
-nmsgs	Suppress creating final error and warning count message.
☆ -opt *n*	Optimize the code in the executable object to the *n*th level. *n* is an optional specifier that must be between 0 and 3. If *n* is omitted, or the entire switch is omitted, optimize to level 3.
-nopt	Suppress optimizing the code in the executable object. This is an obsolete switch; use -opt 0 instead.
-peb	Generate in-line code for the Performance Enhancement Board. This is an obsolete switch; use -cpu peb instead.
☆ -npeb	Suppress creating in-line code for Performance Enhancement Board. This is an obsolete switch; use -cpu any instead.
-slib *pathname*	Treat the input as in include file and produce a precompiled library of include files at program_name.plb or *pathname*.plb.
-subchk	Generate extra subscript checking code in the executable object file. This code signals an error if a subscript is outside the declared range for the array.
☆ -nsubchk	Suppress subscript checking.
☆ -warn	Display warning messages.
-nwarn	Suppress warning messages.
☆ -xrs	Save registers across a call to an external procedure or function.
-nxrs	Do not assume that calls to external routines have saved the registers.

☆ denotes a default option.

The following pages detail each of these options.

6.2.1 -Align and -Nalign: Longword Alignment

The **-align** option is the default.

A standard part of DOMAIN software is the "loader," which loads sections of object modules into memory at runtime. At SR8.0 and later releases, the loader loads each section on a longword boundary, ensuring longword alignment. Longword (32-bit) alignment can enhance performance on DNx60 workstations and those with the M68020 microprocessor. By default, DOMAIN Pascal ensures longword alignment on all workstations, but performance improvements are only seen on DNx60 workstations and those with the M68020 processor. To suppress longword alignment, use the **-nalign** option.

If your target workstation is a DNx60, recompile any object modules that were compiled with an early (pre-SR8.0) version of DOMAIN Pascal.

6.2.2 -B and -Nb: Binary Output

The **-b** option is the default.

If you use the **-b** option, and if your source code compiles with no errors, DOMAIN Pascal creates an object file with the source pathname and the **.bin** suffix. If you specify a pathname as an argument to **-b**, then DOMAIN Pascal creates an object file at pathname.**bin**.

If you use the **-nb** option, DOMAIN Pascal suppresses creating an object file. Consequently, compilation is faster than if you had used the **-b** option. Therefore, **-nb** can be useful when you want to check your source code for grammatical errors, but you don't want to execute it.

Given that error-free DOMAIN Pascal source code is stored in file `test.pas`, here are some sample command lines:

```
$ pas test
{DOMAIN Pascal creates test.bin}

% pas test -b
{DOMAIN Pascal creates test.bin}

$ pas test -b jest
{DOMAIN Pascal creates jest.bin}

B$ pas test -b jest.bin
{DOMAIN Pascal creates jest.bin}

% pas test -nb
{DOMAIN Pascal doesn't create an object file}
```

6.2.3 -Comchk and -Ncomchk: Comment Checking

-Ncomchk is the default.

If you compile with **-ncomchk**, the compiler does not check to see if you've balanced your comment delimiters. Consequently, if you forget to close an open comment, the compiler will probably misinterpret a piece of code as part of a comment. If you compile with the **-ncomchk** option, you get the default results.

If you compile with the **-comchk** option, the compiler checks to see that comment pairs are balanced; that is, that there are no extra left comment delimiters before a right comment delimiter. The left comment delimiters are {, (*, and "; the right comment delimiters are }, *), and ". If you compile with **-comchk**, the compiler returns a warning for every additional left comment delimiter.

For example, the following fragment produces weird results because of an unclosed comment:

```
{This comment should be closed, but I forgot to do it!

x := 0;        {We need this statement.}
```

However, if you compile with **-comchk**, the compiler returns the following warning message:

```
Warning:  Unbalanced comment; another comment start found
before end.
```

Note that **-comchk** causes the compiler to look only for the same kind of left comment delimiter. For example, if you start the comment with (*, the compiler does not flag any extra {.

6.2.4 -Cond and -Ncond: Conditional Compilation

-Ncond is the default.

The **-cond** option invokes conditional compilation. If you compile with **-cond**, DOMAIN Pascal compiles the lines of source code marked with the **%debug** directive. (Refer to the "Compiler Directives" listing in Chapter 4 for details on **%debug**.)

If you compile with **-ncond**, DOMAIN Pascal treats the lines of source code marked with **%debug** as comments.

You can simulate the action of this switch with the **-config** compiler option. For new program development, you should use the **-config** syntax, since this option is considered obsolete.

6.2.5 -Config: Conditional Processing

Use the **-config** option to set conditional variables to true. (Refer to the "Compiler Directives" listing of Chapter 4 for details on the conditional variables.)

You declare these conditional variables with the **%var** compiler directive. By default, their value is false. You can set their value to true with the **%enable** directive (described in the "Compiler Directives" listing) or with the **-config** option. The format of the **-config** option is

-config var1 ... *varN*

where var must be a conditional variable declared with **%var**. For example, consider the following program:

```
PROGRAM config_example;

{ You can use this program to experiment with the }
{ -CONFIG compiler option. }

VAR
    x, y, z : integer16 := 0;


BEGIN
   writeln('The start of the program.');

%VAR first, second, third
%IF first %THEN
   x := 5;
   writeln(x,y,z);
%ENDIF


%IF second %THEN
   y := 10;
   writeln(x,y,z);
%ENDIF


%IF third %THEN
   z := 15;
   writeln(x,y,z);
%ENDIF

   writeln('The end of the program');
END.
```

This program is available on-line and is named config_example.

First, notice what happens when you compile without **-config**.

```
  $ pas config_example
  No errors, no warnings, Pascal Rev n.nn
  $ config_example.bin
  The start of the program.
  The end of the program
```

Now, use the **-config** option to set conditional variables first and third to true. Here's what happens:

```
  $ pas config_example -config first third
  No errors, no warnings, Pascal Rev n.nn
  $ config_example.bin
  The start of the program.
       5         0         0
       5         0        15
  The end of the program
```

To simulate the action of the **-cond** compiler switch, enclose the section of code you want conditionally compiled in an **%if** config_variable **%then** structure. Then use **-config** to set config_variable to true when you want to compile that section of code.

6.2.6 -Cpu: Target Workstation Selection

Use the **-cpu** option to select the target workstations that the compiled program can run on. If you choose an appropriate target workstation, your program might run faster; however, if you choose an inappropriate target workstation, the runtime system will issue an error message telling you that the program cannot execute on this workstation. DOMAIN Pascal can generate code in four possible modes:

- Code that will run on a DSP160, DN460, or DN660 workstation
- Code that will run on a workstation with the M68020 microprocessor and the M68881 floating-point co-processor
- Code that will run on a workstation with a Performance Enhancement Board (PEB)
- Code that will run on any Apollo workstation

You select the code generation mode through the argument that you specify immediately after **-cpu**. Table 6-2 shows the possible arguments and the code generation mode that they select.

Table 6-2. Arguments to the -cpu Option

-cpu 160 -cpu 460 -cpu 660	Generates code for the DSP160, DN460, and DN660 workstations.
-cpu 90 -cpu 330 -cpu 560 -cpu 570 -cpu 580 -cpu 3000	Generates code for workstations with a M68020 processor and a M68881 floating-point unit (includes the DSP90, DN330, DN560, DN570, DN580, and DN3000).
-cpu peb	Generates code for workstations with a PEB (includes the DN100, DN320, DN400, and the DN600, when equipped with an optional PEB).
-cpu any	Generates code for any workstation.

Note that there are many possible arguments to **-cpu**; however, many of them are synonyms. For example, `-cpu 330` produces exactly the same code as `-cpu 560`.

The advantage of compiling with `-cpu any` is that the resulting program can run on any Apollo workstation. This is how Apollo compiles the programs that appear in your */com* or */bin* directories. `-cpu any` is the default.

The advantage of the processor-specific code generation modes is that the compiler generates code optimized for that particular processor, which makes the programs so compiled run faster. The advantage is seen mostly in programs that make heavy use of floating-point. Programs that make heavy use of 32-bit integer multiply and divide might also show significant improvement.

6.2.7 -Db, -Ndb, -Dba, -Dbs: Debugger Preparation

-Db is the default.

Use these switches to prepare the compiled file for debugging by the DOMAIN Language Level Debugger (**debug**). DOMAIN Pascal stores the debugger preparation information within the executable object file, so in general, the more debugger information you request, the longer your executable object file.

If you use the **-ndb** option, the compiler puts no debugger preparation information into the **.bin** file. If you try to debug such a **.bin** file with **debug**, the system reports the following error message:

```
?(debug) The target program has no debugging information.
```

If you use the **-db** option, the compiler puts minimal debugger preparation information into the **.bin** file. This preparation is enough to enter the debugger and set breakpoints, but not enough to access symbols (e.g., variables and constants).

If you use the **-dbs** option, the compiler puts full debugger preparation information into the **.bin** file. This preparation allows you to set breakpoints and access symbols. When you use the **-dbs** option, the compiler sets the **-opt** option. (You can override this with the **-nopt** or **-opt 0** option.)

The **-dba** option is identical to the **-dbs** option except that when you use the **-dba** option, the compiler sets the **-nopt** option (even if you specify **-opt**).

For more complete details on these four options, see the *DOMAIN Language Level Debugger Reference.*

6.2.8 -Exp and -Nexp: Expanded Listing File

-Nexp is the default.

If you compile with the **-exp** option, the compiler generates an expanded listing file. This listing file contains a representation of the generated assembly language code interleaved with the source code.

If you compile with the **-nexp** option, the compiler does not generate a listing file (unless you use the **-map** or **-l** options).

6.2.9 -Idir: Search Alternate Directories For Include Files

The **-idir** option specifies the directories in which the compiler should search for include files if you specify such files using relative, rather than absolute, pathnames. (Absolute pathnames begin with a slash (/), double slash (//), tilde (~), or period (.).)

Without the **-idir** option, DOMAIN Pascal searches for include files in the current working directory. For example, if your working directory is `//nord/minn` and your program includes this directive:

```
%INCLUDE 'mytypes.ins.pas'
```

DOMAIN Pascal searches for that relative pathname at `//nord/minn/mytypes.ins.pas`. However, when you use **-idir**, the compiler first searches for the file in your working directory, and if it doesn't find the file, it looks in the directories you list as **-idir** arguments. When it finds the include file, the search ends. This capability is useful if you have include files stored on multiple nodes or in multiple directories on your node.

For example, consider the following compile command line:

```
$ pas test -idir //ouest/hawaii
```

This command line causes the compiler to search for `mytypes.ins.pas` at `//ouest/hawaii/mytypes.ins.pas` if it can't find `//nord/minn/mytypes.ins.pas`.

You can put up to 63 pathnames following an **-idir** option. Separate each pathname with a space.

6.2.10 -ISO and -NISO: Standard Pascal

The **-niso** option is the default.

DOMAIN Pascal implements a few features differently than ISO standard Pascal. The **-iso** switch lets you tell the compiler to use standard Pascal rules for some of these features.

If you use the option, the compiler flags as an error a **goto** statement that jumps into an **if/then/else** statement. For example, the following is incorrect under the **-iso** switch:

```
label
    bad_jump;
. . .
if num > 0 then
    {statement}
else
bad_jump:                          {WRONG}
    {next statement};
. . .
goto bad_jump;
```

This structure is permitted, although not encouraged, under DOMAIN Pascal.

Compiling with **-iso** also has an effect on comments. With it, comment delimiters no longer are required to match up, which means that the following becomes valid:

```
{ This comment starts with one type of delimiter and ends with another. *)
```

-Iso also tells the compiler to use ISO rules with the **mod** operator. DOMAIN Pascal implements **mod** using the Jensen and Wirth semantics; see the **mod** listing in Chapter 4 for details.

6.2.11 -L and -NL: Listing Files

The **-nl** option is the default.

The **-l** option creates a listing file. The listing file contains the following:

- The source code complete with line numbers. Note that line numbers start at 1 and move up by 1 (even if there is no code at a particular line in the source code). Further note that lines in an include file are numbered separately.
- Compilation statistics.
- A section summary.
- A count of error messages produced during the compilation.

The format for the **-l** option is

-l *pathname*

If you specify a *pathname* following **-l**, the compiler creates the listing file at *pathname*.**lst**. If you omit a *pathname*, the compiler creates the listing file with the same name as the source file. If the source file name includes the **.pas** suffix, **.lst** replaces it. If the source file name does not include **.pas**, **.lst** is appended to the end of the name.

The **-nl** option suppresses the creation of the listing file. (See also **-map** and **-exp.**)

6.2.12 -Map and -Nmap: Symbol Map

The **-nmap** option is the default.

If you use the **-map** option, DOMAIN Pascal creates a map file. A map file contains everything in the listing file (**-l**) plus a special symbolic map. The special symbolic map consists of two sections.

The first section describes all the routines in the compiled file; for example, here is a sample first section:

```
001 EXAMPLE Program(Proc = 00005A,Ecb = 000030,Stack Size = 0)
002 DO_NOTHING Procedure(Proc = 000000,Ecb = 000020,Stack Size = 8)
003 DO_SOMETHING Function(Proc = 00003E,Ecb = 00000C,Stack Size = 8)
```

The preceding data tells you that the compiled file contains a main program (called `example`), a procedure (called `do_nothing`), and a function (called `do_something`). There are three pieces of data inside each pair of parentheses. The first piece tells you the start address in hexadecimal bytes from the start of a section. The second piece is the offset (in bytes) of the routine entry point. The third piece is the size (in hexadecimal bytes) of the stack. The second and third pieces of data probably are of interest to systems programmers only.

For example, in the sample first section, the starting address of the main program was offset 16#5A bytes from the beginning of the PROCEDURE$ section. The offset relative to the beginning of the data section of the main program's routine entry point is 16#30 bytes. Its stack size is 0 bytes.

The second section lists all the variables, types, and constants in the compiled program. For example, here is a sample second section:

```
002 A       Var(-000008/S): CHAR
002 A2      Var(-000006/S): CHAR
001 BI      Type= ARRAY[1..2] OF INTEGER16
001 BILBOA  Const='The rain'
001 Q       Var(+000002/MICROS): CHAR
001 R5      Var(+000004/MICROS): DOUBLE
001 S       Var(+00000C/MICROS): BI
001 X       Var(/MICROS): INTEGER16
001 Y       Var(/D): INTEGER16
003 Z       Var(+000010/S): INTEGER16
```

The map tells you, for example, that R5 is a Var (variable) that is stored +000004 bytes from the beginning of the MICROS section. It also tells you that R5 has the data type **double**. Also, note that /D means the DATA$ section, and /S means the stack.

If you specify **-nmap**, DOMAIN Pascal does not create the special symbol map.

6.2.13 -Msgs and -Nmsgs: Informational Messages

The **-msgs** option is the default.

If you use the **-msgs** option, the compiler produces a final compilation report having the following format:

```
XX errors, YY warnings, Pascal Rev n.nn
```

If you use **-nmsgs**, the compiler suppresses this final report.

6.2.14 -Opt and -Nopt: Optimized Code

The **-opt** option is the default.

If you use the **-opt** option, the compiler optimizes the generated code. If you use the Debugger (described in the "Debug" section of this chapter) to debug an optimized program, you may run into problems because of a fuzzy mapping between source code and generated machine code. See the *DOMAIN Language Level Debugger Reference* for more details.

As part of the **-opt** compiler option, you can specify a predefined level of optimization. The syntax is:

-opt *n*

where *n* is an integer between 0 and 3. The higher the number, the more optimization that will be done. If you omit *n* or omit the **-opt** switch alltogether, the compiler defaults to level 3. If you specify level 0, that is equivalent to **-nopt** and **-nopt** tells the compiler not to optimize the generated code.

Following is a brief description of the optimizations you get at each level between 1 and 3. For a detailed discussion of compiler optimization techniques, consult a general compiler textbook.

-opt 1 Perform limited global common subexpressions and dead code elimination. Transform integer multiplication by a constant into shift and add operations, where appropriate. Perform simple tree transformations, and possibly merge assignment statements.

-opt 2 Perform reaching definitions and global constant folding. (Also includes all of **-opt** 1.)

When the compiler examines code for reaching definitions, it determines whether or not a variable is assigned a value before it uses the value it was last assigned. If there are multiple assignments without intervening uses, the compiler eliminates any unused assignments. For example:

```
x := 1;              {Assignment eliminated because the compiler  }
.                    {reaches the next assignment of x before this}
.                    {value is used.                              }
if a < b then
    x := a
else
    x := b;
```

The constant folding that the compiler performs at this level of **-opt** means that constants are evaluated at compiletime and replaced with their computed values. For instance, `2*3.14159` is replaced by `6.28318`. Likewise, expressions which will always evaluate to true or false are constant folded and conditional code around them eliminated. For example:

```
VAR
    small_range : 0..10;
. . .
if small_range < 0 then       {small_range can never be less than 0   }
    {statements};             {so the compiler constant folds the if  }
                              {expression to false, and generates no  }
                              {code for "statements".                 }
```

-opt 3 Perform live analysis of local variables, redundant assignment statement elimination, global register allocation, and instruction reordering. Also, remove invariant arithmetic expressions from loops, and use the common subexpression (CSE) algorithm to search exhaustively for common subexpressions. **-Opt** 1 makes a limited search for CSEs. (Also includes all of **-opt** 1 and 2.)

Live analysis of local variables involves determining whether the value assigned to a variable in a routine can then be used in that routine. If it can, the variable is considered live; otherwise, it is dead, and there's no need to keep the value in a register. Making optimum use of registers speeds program execution.

Because the compiler does more work at each higher level of optimization, it often takes longer to compile at those higher levels. Therefore, if you are just developing your program, and are compiling to find syntax errors, you might want to compile using a low number. Then, when the program is ready, you can compile with a higher number and so take advantage of all the optimizations.

6.2.15 -Peb and -Npeb: Performance Enhancement

-Npeb is the default.

The **-peb** option is identical to the **-cpu peb** option described earlier in this chapter.

If you specify **-npeb**, DOMAIN Pascal does not generate code optimized for the PEB.

NOTE: For SR8 and later revisions of DOMAIN Pascal, use the **-cpu peb** option instead of the **-peb** option. We do not guarantee support for **-peb** and **-npeb** after SR9.

6.2.16 -Slib: Precompilation of Include Files

Because there usually are a number of common tasks most programs must perform, DOMAIN Pascal programs often contain include files. Frequently used files include */sys/ins/base.ins.pas* and

/sys/ins/error.ins.pas. But it would be time–consuming to compile such files every time a program in which they were included was compiled. The **–slib** option allows you to precompile an include file. Then when you insert that file in your program with the **%slibrary** compiler directive (described in Chapter 4), the compiler knows that the file already has been compiled and doesn't bother to parse it again.

The syntax for **–slib** is:

–slib *pathname*

If you specify a *pathname* following **–slib,** the compiler creates the precompiled file at *pathname*.**plb**. When no *pathname* is present, and the name of the input program file ends in **.pas**, **–slib** replaces that ending with **.plb** and creates the file at program_name.**plb**. If the input filename does not end in **.pas**, **–slib** appends **.plb** to the name.

For example, suppose you want to precompile the file `mystuff.ins.pas`. This command precompiles it and puts the result in `mystuff.ins.plb`.

```
% pas mystuff.ins -slib
```

There are some restrictions on what files can contain if they are going to be precompiled. They can only contain declarations; they may *not* contain routine bodies and may *not* declare variables that would result in the allocation of storage in the default data section, DATA$. This means the declarations must either put variables into a named section, or must use the **extern** variable allocation clause. See Chapter 3 for more information about named sections, and Chapter 7 for details on **extern**.

Any conditional compilation directives in the files you **slib** are executed during precompilation.

If you have several files that you want to combine into a single precompiled library, you can create a container file with a series of **%include** directives. For example, to combine some frequently used include files into the single precompiled library `/sys/ins/domain.plb`, you can create a file `systemstuff.pas` which contains the following:

```
{ Files we often use together. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
```

Then use **–slib** as follows to create the precompiled, combined library.

```
$ pas systemstuff -slib /sys/ins/domain
```

You can include the new precompiled library in any program. For example:

```
PROGRAM errortest;
%SLIBRARY '/sys/ins/domain.plb';

BEGIN
    .
    .
    .
END.
```

6.2.17 –Subchk and –Nsubchk: Subscript Checking

–Nsubchk is the default.

If you use **–subchk,** the compiler generates additional code at every subscript to check that the subscript is within the declared range of the array. This extra code slows your program's execution speed.

If you use **-nsubchk**, the compiler does not generate this extra code.

6.2.18 -Warn and -Nwarn: Warning Messages

-Warn is the default.

If you use **-warn**, the compiler reports all warning messages.

If you use **-nwarn**, the compiler suppresses reporting warning messages (though it does report on the total number of warnings that would have been issued).

We strongly recommend that you avoid using **-nwarn**. Warnings are issued when the compiler believes it knows what the program meant to say, and so thinks it can still generate the right code. But the compiler isn't always right, and if you use **-nwarn**, you won't see the messages that could indicate where the compiler got confused.

6.2.19 -Xrs and -Nxrs: Register Saving

-Xrs is the default.

This option controls whether the compiler believes that the contents of registers are saved across a call to an external routine or a call through a procedure-ptr or function-ptr variable. If you use **-xrs**, the compiler assumes registers are saved, while if you use **-nxrs**, it does not assume registers are saved.

In either case, the compiler always saves register contents when it enters a routine and restores those contents to the registers when it exits the routine.

The primary use for this option is when your program contains calls back to subprograms compiled with pre-SR9.5 compilers. In such a case, you should isolate the portion of your new code that calls the older subprograms, and separately compile that new code with the **-nxrs** option.

6.3 Binding

If your program consists of more than one separately-compiled object file, you must use the binder. If your program consists of only one object file, you do not have to use the binder (though using it causes no harm).

Use the binder utility to combine an object file with other object files to which it refers. The main purpose of the binder is to resolve external references. An external reference is a symbol (i.e., variable, constant, or routine) that you refer to in one object file and define in another. To invoke the binder, enter a command line of the following format:

$ **bind** pathname1...*pathnameN* *option1*...*optionN*

A pathname must be the pathname of an object file (created by a compiler) or a library file (created by the librarian). Your bind command line must contain at least one pathname.

Your bind command line can also contain zero or more binder *option*s, the most important of which is **-b**. If you use the **-b** option, the binder generates an executable object file. If you forget to use the **-b** option, the binder won't generate an output object file.

For example, suppose you write a program consisting of three source code files -- TEST_MAIN.PAS, MOD1.PAS, and MOD2.PAS. To compile the source code, you issue the following three commands:

```
$ pas TEST_MAIN
$ pas MOD1
$ pas MOD2
```

The DOMAIN Pascal compiler creates TEST_MAIN.BIN, MOD1.BIN, and MOD2.BIN. To create an executable object, bind the three together with a command like the following:

```
$ bind TEST_MAIN.BIN  MOD1.BIN  MOD2.BIN  -B T3
```

This command creates an executable object file in filename T3.

Refer to the *DOMAIN Binder and Librarian Reference* for a complete discussion of the binder and its options.

6.4 Using Libraries

As mentioned in the previous section, you can put library files on the bind command line. You create library files with the DOMAIN Librarian utility which is detailed in the *DOMAIN Binder and Librarian Reference*.

Actually, the DOMAIN system supports several different kinds of libraries. In addition to library files, the DOMAIN system also supports user-defined installed libraries, system-defined installed libraries, system-defined global libraries, and the user-defined global library. All are detailed in the *DOMAIN Binder and Librarian Reference*.

On some operating systems, you must bind language libraries and system libraries with your own object files. On the DOMAIN system, there is no need to do this as the loader binds them automatically when you execute the program.

6.5 Executing the Program

To execute a program, simply enter its pathname. For example, to execute program T3, just enter

```
$ T3
```

The operating system searches for a file named T3 according to its usual search rules, then calls the loader utility. The loader utility is user transparent. It binds unresolved external symbols in your executable object file with global symbols in the language and system libraries. Then, it executes the program.

By default, standard input and standard output for the program are directed to the keyboard and display. You can redirect these files by using the Shell's redirection notation (described in the *DOMAIN System User's Guide*). For example, to redirect standard input when you invoke T3, type

```
$ T3  <TRADING_DATA
```

The character "<" redirects standard input to the file TRADING_DATA. You can redirect standard output in a similar fashion, for example:

```
$ T3  >results
```

This command uses the character ">" to redirect standard output for T3 to the file results.

6.6 Debugging the Program

The DOMAIN system supports three tools to help you debug a DOMAIN Pascal program -- **debug**, **tb**, and **crefpas**.

6.6.1 Debug

You can use the language level debugger (**debug**) utility to debug your program. This utility allows you to set breakpoints so you can step through a program examining variables and finding bugs. In order to fully debug your program, you must use the **-dba** or **-dbs** options when you compile. (We recommend **-dbs.**) You don't have to use any special options when you bind. To invoke the debugger, simply enter a command of the format:

```
$ debug object_file
```

where object_file is the name of an executable object file. For example:

```
$ debug T3
```

The *DOMAIN Language Level Debugger Reference* details the debugger.

6.6.2 Traceback (tb)

If you execute the program and the system reports an error, you can use the **tb (traceback)** utility to find out what routine triggered the error. To invoke **tb**, enter the command

```
$ tb
```

immediately after a faulty execution of the program.

For example, suppose you invoke T3, encounter an error, and then invoke **tb**. The whole sequence might look like the following:

```
$ T3
Enter a real number -- x
?(sh) "./t1.bin" - invalid read data (library/Pascal)
In routine "ERROR" line 366.
$ TB
invalid read data (from library / Pascal)
In routine "ERROR" line 366
Called from "PAS_$READ" line 1602
Called from "SAMPLE" line 11
```

Tb first reports the error, which in this case is

```
invalid read data (from library / Pascal)
```

Then, **tb** shows the chain of calls leading from the routine in which the error occurred all the way back to the main program block. For example, routine ERROR reported the error. ERROR was called by the PAS_$READ routine. And, PAS_$READ was called by line 11 of the SAMPLE routine. Since ERROR and PAS_$READ are system routines, it is probable that the error occurred at line 11 of the SAMPLE routine.

The *DOMAIN System Command Reference* details the **tb** utility.

6.6.3 Crefpas

Crefpas produces a cross-reference list of the identifiers (other than DOMAIN Pascal reserved words) used in a DOMAIN Pascal source program. The listing is in four parts, each part beginning on a new page. The first part lists the program, with line numbers added. The second part lists **begin, repeat,** and **case** statements, with their associated ending statement lines identified. The third part lists **begin, repeat,** and **case** statements and their associated ending statements, but with the ending statements listed first. The fourth part lists all identifiers (other than DOMAIN Pascal reserved words) and the line numbers on which they appear. Appearances in declaration sections are flagged with a 'D'.

Insert files are not included in the cross-reference listing.

To invoke **crefpas**, enter a command of the following format:

```
% crefpas pathname -l list_file
```

For pathname, enter the Pascal source file to be cross-referenced.

By default, **crefpas** sends output to pathname.lst. If pathname ends in **.pas**, then **.lst** replaces **.pas**. However, you can redirect the output with the *-l list_file* option. (**Crefpas** automatically appends the **.lst** suffix to *list_file* if it is absent.)

Now consider an example. Suppose a program stored in source code file `TEST.PAS` has a **begin** statement on line 20, a **repeat** statement on line 25, a **case** statement on line 30, and the associated ending statements on lines 50, 40, and 35, respectively. The command

```
% crefpas TEST -l list_it
```

produces the following analysis in the file `list_it.lst`:

```
Part 1

   (The normal program listing, with line numbers added.)

Part 2

        20>B>50,#####25>R>40#####30>C>35

Part 3

        35<C<30#####40<R<25#####50<B<20

Part 4

    (The list of all identifiers with their line numbers.)
```

6.7 The DSEE Product

The DSEE (DOMAIN Software Engineering Environment) package is a support environment for software development. DSEE helps engineers develop, manage, and maintain software projects; it is especially useful for large-scale projects involving a number of modules and developers. You can use DSEE for:

- Source code and documentation storage and control
- Audit trails
- Release and Engineering Change Order (ECO) control
- Project histories
- Dependency tracking
- System building

This chapter described a traditional program development cycle (i.e., compiling, building libraries, binding, debugging); the DSEE product provides some sophisticated enhancements to this cycle. For information on the optional DSEE product, see the *DOMAIN Software Engineering Environment (DSEE) Reference*.

6.8 DOMAIN/Dialogue

DOMAIN/Dialogue is a tool for defining the interface to an application program and specifying how the interface should be presented to users of the application. The primary advantage of DOMAIN/Dialogue is that it lets you create interfaces separately from the application code. For the user interface, DOMAIN/Dialogue lets you:

- Focus more time and attention on the interface than is possible when it is intertwined with the application code.
- Develop modular interfaces that are consistent in design from application to application because they are developed with the same set of tools.
- Use an iterative approach to interface design. A program's user interface can be rapidly prototyped and modified without affecting the application code. Successive testing and refinement are relatively easy, making it possible to fine–tune the interface.
- Develop multiple user interfaces to a program, allowing users to choose a style of interaction with which they feel most comfortable.

For the application, DOMAIN/Dialogue enables you to:

- Write less code. Because DOMAIN/Dialogue handles interactions with the user, the application designer does not have to provide the code for doing so.
- Achieve a modular approach to writing code that promotes phased and iterative application development independent of user interface development.

For details about DOMAIN/Dialogue, see the *DOMAIN/Dialogue User's Guide*.

Chapter 7

External Routines and Cross-Language Communication

This chapter describes how to create and call Pascal modules and how to call FORTRAN and C routines from a DOMAIN Pascal program. Briefly, this chapter covers the following topics:

- Creating Pascal modules
- Accessing a Pascal variable or routine stored in a separately compiled module
- Accessing FORTRAN routines from a Pascal program
- Accessing C routines from a Pascal program

7.1 Modules

It is usually a good idea to break a large Pascal program into several separately compiled modules. After you compile each module, you can bind the resulting object files into one executable object file. (See Chapter 6 for details about binding.)

Every program must consist of one main program and zero or more modules. Chapter 2 contains a description of the main program's format. A main program must begin with the keyword **program**. A module begins with the keyword **module**. It takes the format shown in Figure 7-1.

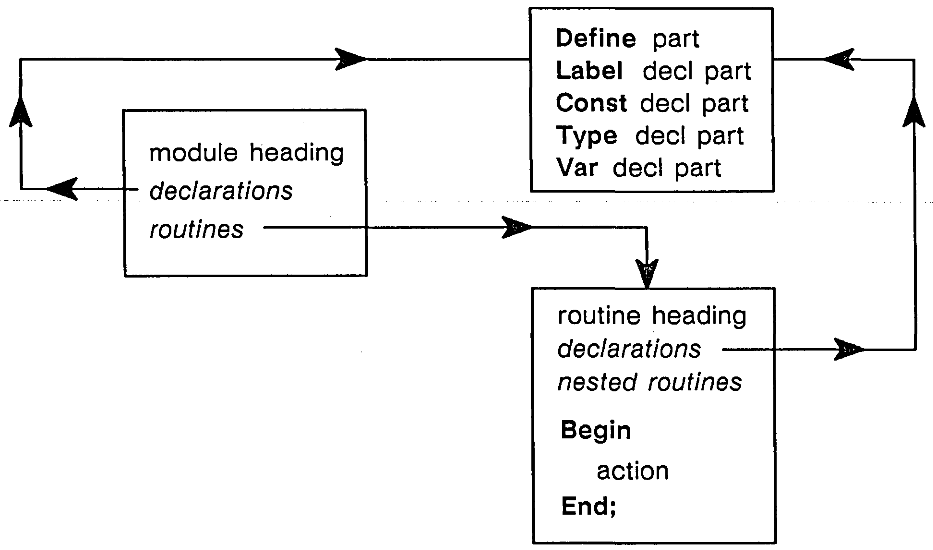

Figure 7-1. Format of a Module

At runtime, the start address of the program is the first statement in the main routine of the main program.

The format of a module is very similar to the format of the main program shown in Figure 2-1. The differences between the formats are:

- A module takes a module heading rather than a program heading. (See the next section for a description of the module heading.)
- A module can contain zero or more routines; each routine must have a name. That is, the main program always contains one main (unnamed) routine, but a module must consist of named routines only.
- The declarations part of a module may contain a **define** part. The "Method 2" section of this chapter describes the **define** part.

7.1.1 Module Heading

The module heading is similar to a program heading except that it starts with the keyword **module** instead of **program**, and that it cannot take a file_list. Therefore, the module heading takes the following format:

module name, *code_section_name, data_section_name*;

Name must be an identifier. The name you pick has no effect on the module.

Code_section_name and *data_section_name* are optional elements of the module heading. Use them to specify nondefault section names for the code and data in the module. A section is a named contiguous area of memory that shares the same attributes. (See the *DOMAIN Binder and Librarian Reference* for details on sections and attributes.) By default, DOMAIN Pascal assigns all the code in your module to the PROCEDURE$ section and all the data in your module to the DATA$ section. To assign your code and data to nondefault sections, specify a *code_section_name* and a *data_section_name*.

Chapter 2 described the format of a main program, this chapter describes the format of a module, and Chapter 6 detailed the method for compiling and binding modules and main programs. The following sections explain how to write your source code so that the separately compiled units can communicate with one another.

7.2 Accessing a Variable Stored in Another Pascal Module

DOMAIN Pascal provides four methods for accessing the value of a variable stored in a separately compiled file. The trick to the first three methods is using the correct *variable_allocation_clause* when declaring variables. The optional *variable_allocation_clause* precedes the data type in a **var** declaration part as shown below:

```
var (section_name)
   identifier_list1 : variable_allocation_clause data_type1;
                      .
                      .
                      .
   identifier_listN : variable_allocation_clauseN data_typeN;
```

For *variable_allocation_clause*, you must enter one of the following three identifiers:

- **Define** -- tells Pascal to allocate the variable in a static data area and make its name externally accessible.
- **Extern** -- tells Pascal to not allocate the variable because it is possibly allocated in a separately compiled module or program.
- **Static** -- tells Pascal to allocate the variable in a static data area and keep its name local to this module or program. Use the **static** clause when a variable needs to retain its value from one execution of a routine to the next. For example, the following fragment declares x as a statically allocated variable:

  ```
  VAR
     x : static integer16;
  ```

  After the execution of the routine in which x is declared, x still retains its value.

The following sections demonstrate three of the four methods for accessing a variable stored in another DOMAIN Pascal module.

7.2.1 Method 1

The following fragments demonstrate the first method for accessing an externally stored variable:

```
program Math;                        module Sub2;
var                                  var
   x : EXTERN integer16;                x : DEFINE integer16 := 8;

                                     Procedure twoa;
BEGIN                                BEGIN
   writeln(x); {access x}               writeln(x); {access x}
END.                                 END;
```

This method uses the variable allocation clauses **extern** and **define** to make the value of x available to both Math and Sub2. The **extern** clause in Math tells the compiler that variable x is probably defined somewhere outside of Math. The **define** clause in Sub2 tells the compiler that x is defined within Sub2; the ":= 8" tells the compiler to initialize x to 8. If x is not initialized, the writeln(x) statement generates totally undefined output.

When you bind, the binder matches the external reference to x in program Math with the global symbol x defined in module Sub2. Note that you can specify x as an **extern** in as many modules as you want; however, you should only **define** x in one module. If you **define** x in more than one module, the binder reports an error.

7.2.2 Method 2

The following fragments demonstrate the second method for accessing an externally stored variable:

```
program Math;                       module Sub2;
var                                 var
   x : EXTERN integer16;               x : EXTERN integer16;
                                    DEFINE
                                       x := 8;

                                    Procedure twoa;
BEGIN                               BEGIN
   writeln(x); {access x}              writeln(x); {access x}
END.                                END;
```

Method 2 introduces the **define** statement. The **define** statement is similar to the **define** variable allocation clause. The **define** statement in Sub2 serves two purposes. First, it tells the compiler that x is defined in module Sub2. Second, it tells the compiler that the initial value of x is 8.

The **define** statement takes the following syntax:

define
  variable1 *:= initial_value1*;

  .
  .
  .

  variableN := initial_valueN;

Notice that the *initial_value* is optional. If you do not provide an *initial_value*, the value of variable is totally undefined; that is, there is no way to predict what the variable's initial value will be.

NOTE: Order is crucial. If you use Method 2, the **var** declaration part must precede the **define** statement. (However, see Method 3 later in this chapter.)

Putting a **define** statement in module Sub2 may appear contradictory at first. After all, a **define** statement means "it's defined here" and an **extern** clause implies "it's defined somewhere else." It seems strange, but this apparent contradiction can actually simplify programming. For example, suppose the **extern** clause is stored in an **%include** file. When writing an **%include** file, there is no way to tell where it's going to end up. It may just end up in a file with a matching **define** statement. If it does, no harm is done.

7.2.2.1 Initializing Extern Variable Arrays

The "Initializing Variable Arrays" section in Chapter 3 contained a lengthy section on initializing variable arrays. The following paragraphs explain how to initialize an array variable that has the **extern** variable allocation clause.

Ordinarily the **–subchk** compiler option causes the compiler to generate code to check that subscripts are within the defined range of an array. However, if you do not specify initialization data for an **extern** array variable, the compiler does not generate this extra code when you compile with **–subchk**. The compiler cannot check subscripts because the upper bound is not known. If there is data initialization with the **extern** declaration, the data is ignored until the variable is defined in a file. However, since the upper bound is known, the compiler can perform subscript checking.

When you let the compiler determine the upper bound of the array, make sure that variable declarations do not share the same type declaration. If they do, the first data initialization for that variable sets the size for all other variables. All initialized variables are then checked against that size, just as if you had specified a constant upper bound.

For example, consider the following fragment:

```
VAR
   table1, table2  : EXTERN array[1..*] of char;
   table3          : EXTERN array[1..*] of char;
   table4          : EXTERN array[1..*] of char;

DEFINE
   table1 := 'This table sets the size,';
   table2 := 'so this table will be truncated';
   table3 := 'But separate variable declarations';
   table4 := 'solve the problem.';
```

In the preceding example, variables `table1` and `table2` share the same anonymous type declaration. Since `table1` precedes `table2`, DOMAIN Pascal uses the defined size of `table1` to set the size for `table2`. In this case, `table1` is a 26-character string. Therefore, the compiler truncates the string `"so this table will be truncated"` to its first 26 characters. Variables `table3` and `table4` are declared separately, so their initializations do not affect each other.

For arrays with a base type of **char**, DOMAIN Pascal issues a warning message when it encounters truncation (as for `table2`). However, if the array has a base type other than **char**, the compiler issues an error message when it encounters truncation.

7.2.3 Method 3

The following fragments demonstrate the third method for accessing an externally stored variable:

```
program Math;                       module Sub2;
VAR                                 DEFINE
   x : EXTERN integer16;               x;
                                    VAR
                                       x : EXTERN integer16 := 8;

                                    Procedure twoa;
BEGIN                               BEGIN
   writeln(x); {access x}              writeln(x); {access x}
END.                                END;
```

Method 3 is similar to Method 2 except that the initialization (:= 8) is done in the **var** declaration part rather than in the **define** statement.

> **NOTE:** Order is crucial. If you use Method 3, the **define** statement must precede the **var** declaration part.

7.2.4 Method 4

The following fragment demonstrates this fourth method for accessing an externally stored variable:

```
Program Math;                               Module Sub2;
VAR (my_sec)                                VAR (my_sec)
   x : integer16 := 5;                         x : integer16;
   y : real := 6.2;                            y : real;
   q : array[1..3] of char := 'cat';           q : array[1..3] of char;

                                            Procedure twoa;
BEGIN                                       BEGIN
   writeln(x, y, z);                            writeln(x, y, z);
END.                                        END;
```

Notice that the variables in each module occupy the same nondefault section name, my_sec. This means that at runtime, the variables x, y, and q from Math occupy the same memory locations as x, y, and q from Sub2. Therefore, whatever is assigned to x in Math, can be retrieved in Sub2, and vice versa.

There's no requirement that the variables in Sub2 have the same names as the variables in Math. However, it does make your program far easier to understand if you do give them the same names. Taking this philosophy one step further, you could greatly simplify variable declaration by putting a named **var** declaration part into a separate file and using **%include** to put it into every module.

You should preserve the order of declarations from one module to the next. For example, suppose the variable declarations look like this:

```
Program Math;                               Module Sub2;
VAR (my_sec)                                VAR (my_sec)
   x : integer16 := 5;                         y : real;
   y : real := 6.2;                            x : integer16;
   q : array[1..3] of char := 'cat';           q : array[1..3] of char;
```

If you try to access x or y in Sub2, you get garbage results at runtime. That's because when compiling Sub2, the compiler sees y as being the first four bytes in my_sec. However, when compiling Math, the compiler sees y as being bytes 2 through 5 in my_sec.

7.3 Accessing a Routine Stored in Another Pascal Module

DOMAIN Pascal provides two methods for accessing a procedure or function stored in a different module or program. This section explains both methods, but first describes the **extern** and **internal** routine options, which are both critical to that description.

7.3.1 Extern

The routine option **extern** serves exactly the same purpose for routines that the variable allocation clause **extern** serves for variables. Namely, it specifies that a routine is possibly defined in a separately compiled file. By specifying a routine as **extern**, you can call it even if it is defined in another file.

Chapter 5 describes the format of routine options. **Extern** is like any other routine option, except that when you use **extern**, you put the routine heading at the end of the main declaration part of the program or module.

7.3.2 Internal

By default, all routine names in modules are global symbols, and all routine names in main programs are local symbols. A routine name from the main program can never become a global symbol. However, you can make a routine name from a module into a local symbol by specifying the routine option **internal**. See the "Routine Options" section in Chapter 5 for details on the syntax rules of routine options.

7.3.3 Method A

Figure 7-2 demonstrates the first method for accessing an externally stored routine.

In Figure 7-2, program `math` refers to function `exponent`; however, function `exponent` is stored in module `math2`. Therefore, function `exponent` is declared as an **extern** in program `math`. There is no need to do anything special to module `math2` because the compiler automatically makes `exponent` a global symbol in `math2`. At bind time, the binder resolves external symbol `exponent` with global symbol `exponent`.

If you want to keep `exponent` local to module `math2`, you can make the following change to the source code:

```
change    -->  FUNCTION exponent(number, power : INTEGER16) : REAL;
to        -->  FUNCTION exponent(number, power : INTEGER16) : REAL; INTERNAL;
```

```
PROGRAM math;

VAR
     n, p : integer16;
     answer : real;

FUNCTION exponent(n,p : integer16) : real; EXTERN;
{exponent is defined outside this file.}

BEGIN
    writeln('The main program calls a separately-compiled routine in');
    writeln('order to calculate the value of an integer raised to an');
    writeln('integer power.');
    writeln;
    write('Enter an integer -- ');       readln(n);
    write('Raised to what power -- '); readln(p);
    answer := exponent(n,p);
    writeln(n:1, ' raised to the ', p:1, ' power is ', answer:1);
END.
```

```
MODULE math2;

{ You must bind this module with program math. }

VAR
    number, power, count : INTEGER16;
    run : INTEGER32;

FUNCTION exponent(number, power : INTEGER16) : REAL;
BEGIN
    writeln('This is mod 2');
    if power = 0 then exponent := 1;
    run := 1;
    for count := 1 to abs(power) do
          run := run * number;
    if power > 0
        then exponent := run
        else exponent := (1 / run);
END;
```

Figure 7–2. Method A for Accessing an External Routine

7.3.4 Method B

Figure 7–3 demonstrates the second method for accessing an externally-stored routine.

```
PROGRAM math;

VAR
     n, p : integer16;
     answer : real;

FUNCTION exponent(n,p : integer16) : real; EXTERN;
{exponent is defined outside this file.}

BEGIN
    writeln('The main program calls a separately-compiled routine in');
    writeln('order to calculate the value of an integer raised to an');
    writeln('integer power.');
    writeln;
    write('Enter an integer -- ');      readln(n);
    write('Raised to what power -- '); readln(p);
    answer := exponent(n,p);
    writeln(n:1, ' raised to the ', p:1, ' power is ', answer:1);
END.
```

```
MODULE math2;

{ You must bind this module with program math. }

DEFINE
    exponent;

FUNCTION exponent(number, power : INTEGER16) : REAL; EXTERN;

VAR
    number, power, count : INTEGER16;
    run : INTEGER32;


FUNCTION exponent;
BEGIN
    writeln('This is mod 2');
    if power = 0 then exponent := 1;
    run := 1;
    for count := 1 to abs(power) do
          run := run * number;
    if power > 0
        then exponent := run
        else exponent := (1 / run);
END;
```

Figure 7–3. Method B for Accessing an External Routine

Under Method B you call external routines exactly as you do in Method A. Therefore, program `math` is unchanged from Method A. However, you must make the following three changes in module `math2`:

- `DEFINE exponent;` -- Use a **define** statement to tell the compiler that `exponent` is a global symbol defined in module `math2`. With one exception, **define** takes the syntax described in the "Accessing a Variable Stored in Another Pascal Module" section earlier in this chapter. The one exception is that you cannot assign an initial value to a procedure or function symbol.

- `FUNCTION exp(number, power : INTEGER16) : REAL; EXTERN;` -- Copy the function declaration from program `math`. Since this line should be an exact copy of the function declaration in the calling module, you should put this line into a separate file and **%include** it into both the module and the program. That way, you only keep one version of the function declaration.

- `FUNCTION exponent;` -- Notice that there are no arguments here; only the name of the function.

Figure 7-4 consists of one main program and two modules. The main program calls two external routines stored in the two modules. The two modules access some variables that the main program defines.

```
Program relativity1;

CONST
   speed_of_light = 2.997925e8;

{ Both of the following routines are defined in another file: }
Procedure convert_mph_to_light(IN speed_in_mph  : real;
                               OUT fraction_of_c : real); EXTERN;
Function contracted_length(IN rest_length : real;
                           IN fraction_of_c: real) : real; EXTERN;

VAR
   speed_in_mph, fraction_of_c, rest_length : real;
   c : DEFINE real := speed_of_light;  {Make the value of c globally known.}

BEGIN
   write('Enter the speed of the object (in mph) -- ');
   readln(speed_in_mph);
   convert_mph_to_light(speed_in_mph, fraction_of_c);

   write('What is the rest length of the object (in meters) -- ');
   readln(rest_length);
   write('It will be perceived in the rest frame of reference as ');
   writeln(contracted_length(rest_length, fraction_of_c):8:6, ' meters');
END.
```

Figure 7-4. Another Example of Calling External Routines

```
Module relativity2;
DEFINE
    convert_mph_to_light;

Procedure convert_mph_to_light(IN speed_in_mph   : real;
                               OUT fraction_of_c : real); EXTERN;

VAR
    speed_in_mph, speed_in_mps, fraction_of_c, percentage : real;
    c : EXTERN real;

{ The following procedure converts a speed (given in miles per hour) }
{ into a percentage of the speed of light.                           }
Procedure convert_mph_to_light;
BEGIN
   speed_in_mps := (0.44444 * speed_in_mph);
   fraction_of_c := speed_in_mps / c;
   percentage := (100 * fraction_of_c);
   writeln('This speed is ', percentage:8:6, ' percent of c.');
END;
```

```
Module relativity3;

DEFINE
   contracted_length;

Function contracted_length(IN rest_length   : real;
                           IN fraction_of_c : real) : real; EXTERN;

VAR
   rest_length : real;
   speed_in_mps : real;

{ This function calculates the relativistic length contraction. }
Function contracted_length;
BEGIN
   contracted_length := rest_length * sqrt(1 - sqr(fraction_of_c) );
END;
```

Figure 7-4. Another Example of Calling External Routines (Continued)

Suppose you store program relativity1 in file relativity1.pas, module relativity2 in file relativity2.pas, and module relativity3 in file relativity3.pas. To compile the three files, enter the following three commands:

```
$  pas relativity1
$  pas relativity2
$  pas relativity3
```

You must now bind them together by entering a command like the following:

```
$  bind relativity.bin relativity2.bin relativity3.bin -b einstein
```

Here is a sample execution of the program:

```
$ einstein
Enter the speed of the object (in mph) -- 4500000
This speed is 0.667121 percent of c.
What is the rest length of the object (in meters) -- 10
It will be perceived in the rest frame of reference as 9.999778 meters
```

7.4 Calling a FORTRAN Routine From Pascal

DOMAIN Pascal permits you to call routines written in DOMAIN FORTRAN source code. To accomplish this, perform the following steps:

1. Write source code in DOMAIN Pascal that refers to an external routine. Compile with the DOMAIN Pascal compiler. DOMAIN Pascal creates an object file.

2. Write source code in DOMAIN FORTRAN. Compile with the DOMAIN FORTRAN compiler. DOMAIN FORTRAN creates an object file.

3. Bind the object file(s) the Pascal compiler created with the object file(s) the FORTRAN compiler created. The binder creates one executable object file.

4. Execute the object file as you would execute any other object file.

The following sections describes steps 1 and 2. For information on steps 3 and 4, see Chapter 6.

NOTE: The following sections explain how to call DOMAIN FORTRAN from DOMAIN Pascal. If you want to learn how to call Pascal from FORTRAN, see the *DOMAIN FORTRAN Language Reference.*

7.5 Data Type Correspondence for Pascal and FORTRAN

There is really no difference between making a call to a FORTRAN function or subroutine and making a call to an **extern** Pascal routine. However, before passing data between DOMAIN Pascal and DOMAIN FORTRAN, you must understand how Pascal data types correspond to FORTRAN data types. Table 7-1 lists these correspondences.

Table 7-1. DOMAIN Pascal and DOMAIN FORTRAN Data Types

DOMAIN Pascal	DOMAIN FORTRAN
Integer, Integer16	**Integer*2**
Integer32	**Integer, Integer*4**
Real, Single	**Real, Real*4**
Double	**Double Precision, Real*8**
Char	**Character*1**
Boolean record	**Logical**
Set	set emulation calls
user-declared record	**Complex**
Array	array (with restrictions)
Pointer	**Pointer** statement

The integer, real, and character data types in both languages correspond very well to each other. For example, Pascal's **integer16** data type is identical to FORTRAN's **integer*2** data type, and a **real** in one language is exactly the same as a **real** in the other.

There is a difference in what the keyword **integer** means in the two languages. **Integer** in DOMAIN Pascal is a 2-byte entity, while in DOMAIN FORTRAN, **integer** is four bytes by default. To avoid any confusion, you should use the specific integer data types (**integer16**, **integer32**, **integer*2**, and **integer*4**) rather than the generic **integer**.

Unlike Pascal, DOMAIN FORTRAN doesn't actually support a set data type. However, you can make special set emulation calls from within a DOMAIN FORTRAN program. Therefore, you can pass a set variable as an argument from a Pascal program and use the set emulation calls within your FORTRAN program.

7.5.1 Boolean and Logical Correspondence

Pascal's **boolean** data type and FORTRAN's **logical** data type serve identical purposes, namely, to hold a value of true or false. However, **boolean** takes up one byte of main memory while **logical** takes up four bytes with the first byte being where the true or false value is set. To make the data types match up, you should create a record type in Pascal that contains one **boolean** for the "real" value, and three place-holding **booleans**. That is:

```
TYPE
   boolrec = record
                real_bool : boolean;
                a,b,c     : boolean;
             end;
```

7.5.2 Simulating FORTRAN's Complex Data Type

Unlike FORTRAN, DOMAIN Pascal doesn't support a predeclared **complex** data type. However, you can easily declare a Pascal record type that emulates **complex**. In FORTRAN, **complex** consists of two single-precision real numbers. Therefore, you could define a **complex** Pascal record as follows:

```
TYPE
   complex = record
                r         : single;
                imaginary : single;
             end;
```

7.5.3 Array Correspondence

Single-dimensional arrays (including **boolean/logical** arrays when you make the adjustments described above) of the two languages correspond perfectly; for example:

In DOMAIN Pascal	*In DOMAIN FORTRAN*
`x : Array[1..10] of CHAR`	`character*10   x`
`x : Array[1..50] of INTEGER16`	`integer*2  x(50)`
`x : Array[1..20] of DOUBLE`	`real*8     x(20)`

The one exception is that you cannot declare a DOMAIN FORTRAN **char** array of unspecified length as a parameter. For example, do not specify an array like the following as a parameter in the DOMAIN FORTRAN program:

```
CHARACTER*(*)    x
```

Multidimensional arrays in the two languages do not correspond very well. The tricky part is that Pascal represents multidimensional arrays differently than FORTRAN. To represent arrays in DOMAIN Pascal, the right-most element varies fastest. For example, DOMAIN Pascal represents the six elements of an `array[1..2, 1..3]` in the following order:

```
1,1
1,2
1,3
2,1
2,2
2,3
```

However, the left-most element varies fastest in DOMAIN FORTRAN arrays. Therefore, DOMAIN FORTRAN represents the six elements of an `array(2,3)` in the following order:

```
1,1
2,1
1,2
2,2
1,3
2,3
```

Obviously this can lead to confusion if you pass a multidimensional Pascal array as an argument to a DOMAIN FORTRAN parameter. However, there is a way to avoid this confusion. Simply declare the array dimensions of the DOMAIN FORTRAN parameter in reverse order. For example, instead of declaring `integer*4 array(2,3)`, declare `integer*4 array(3,2)`. Following are two more examples:

Argument In Pascal	*Parameter In FORTRAN*
`x : array[1..5, 1..10] of real`	`real*4  x(10,5)`
`x : array[1..2, 1..3, 1..4] of real`	`real*4  x(4,3,2)`

7.6 Passing Data Between FORTRAN and Pascal

There are two ways to pass data between a DOMAIN Pascal program and a DOMAIN FORTRAN function or subroutine. You can either establish a common section of memory for sharing data, or you can pass the data as an argument to a routine. The next section demonstrates the second method, while the following paragraphs demonstrate the first method.

Earlier in this chapter you learned how to use a named section to pass data between two separately compiled Pascal modules. A named section in DOMAIN Pascal is identical to the named **common** area of DOMAIN FORTRAN. If you give a named section the same name as a named **common** area, the binder establishes a section of memory for sharing data.

For example, suppose that you want both a Pascal program and a FORTRAN function or subroutine to access two variables -- a 16-bit integer and an 8-byte (double-precision) real number. In the DOMAIN Pascal program, you can declare the two variables as follows:

```
VAR (my_sec)
    x : INTEGER16;
    d : DOUBLE;
```

If you want the value of these two variables to be accessible from the DOMAIN FORTRAN program, declare them as follows in the FORTRAN program:

```
INTEGER*2  x
REAL*8     d
COMMON /my_sec/ x,d
```

Remember to preserve the same order of variable declaration in the **common** statement that you did in the **var** declaration part. For example, you will get peculiar runtime results if you declare your **common** statement as

```
COMMON /my_sec/ d,x
```

7.7 Calling FORTRAN Functions and Subroutines

This section demonstrates how to call a DOMAIN FORTRAN function or subroutine from a DOMAIN Pascal program. Calling DOMAIN FORTRAN from DOMAIN Pascal is trivial; the only possible complication is that the data types of the arguments and parameters may not correspond perfectly. (See the "Data Type Correspondence" section earlier in this chapter for ways to remedy the data type mismatches.)

DOMAIN Pascal supports a variety of parameter types including value parameters, variable parameters, **in**, **out**, and **in out** parameters. DOMAIN FORTRAN supports only one kind of parameter type, and it is equivalent to the **in out** parameter type in DOMAIN Pascal. That is, DOMAIN FORTRAN accepts whatever value(s) you pass to it, and in turn, always passes a value back.

7.7.1 Calling a Function

The following shows a DOMAIN Pascal program that calls the DOMAIN FORTRAN function listed after it. The call is trivial since DOMAIN Pascal's **single** data type corresponds perfectly to the **real*4** data type of DOMAIN FORTRAN.

```
program pascal_fortran1;
{ This program calls an external function named HYPOTENUSE. Although it will }
{ end up calling a FORTRAN version of that function, this same source code   }
{ could call a Pascal version of HYPOTENUSE.                                  }
VAR
    leg1, leg2 : single;

function hypotenuse (in out leg1, leg2 : single) : single; extern;

BEGIN

writeln ('This program calculates the hypotenuse of a right');
writeln ('triangle given the length of its two legs.');
write ('Enter the length of the first leg -- ');
readln (leg1);
write ('Enter the length of the second leg -- ');
readln (leg2);

writeln ('The length of the hypotenuse is: ', hypotenuse(leg1,leg2):5:2);

END.

*********************************************************************
* This is a FORTRAN function for calculating the hypotenuse of a
* right triangle. You don't have to do anything special to this file
* to make it callable by Pascal. In fact, this function could just
* as easily be called by a FORTRAN program.

      real*4 function hypotenuse(l1, l2)
      real*4 l1, l2   {real*4 corresponds to the Pascal data type single.}
      hypotenuse = sqrt((l1 * l1) + (l2 * l2))
      end
```

These programs are available on-line and are named pascal_fortran1 and hypotenuse.

7.7.2 Calling a Subroutine

A function in Pascal corresponds to a function in FORTRAN. A procedure in Pascal corresponds to a subroutine in FORTRAN. In the following example, HYPOTENUSE changes from a function to a subroutine and the Pascal program changes to reflect that it is expecting an external procedure. Note that the Pascal program could actually be calling a Pascal procedure. There's nothing in the program that designates the language in which the called procedure is written.

```
program pascal_fortran2;

VAR
    leg1, leg2, result : real;

procedure hypotenuse (in out leg1, leg2, result : real) ; extern;

BEGIN

writeln ('This program calculates the hypotenuse of a right');
writeln ('triangle given the length of its two legs.');
write ('Enter the length of the first leg -- ');
readln (leg1);
write ('Enter the length of the second leg -- ');
readln (leg2);

hypotenuse(leg1,leg2,result);
writeln ('The length of the hypotenuse is: ', result:5:2);

END.

*******************************************************************
* This is a FORTRAN subroutine for calculating the hypotenuse of a
* right triangle.

      subroutine hypotenuse(l1, l2, result)
      real*4 l1, l2, result

      result = sqrt((l1 * l1) + (l2 * l2))
      end
```

These programs are available on-line and are named pascal_fortran2 and hypot_sub.

7.7.3 Passing Character Arguments

Passing arguments when two languages' data types match exactly is relatively easy, but passing them when they don't often means you need to do extra work.

If you pass a string of **chars** from DOMAIN Pascal to DOMAIN FORTRAN, you should add an extra parameter to the Pascal routine heading. This is because FORTRAN adds an implicit string length argument whenever it passes a **character** string back to a calling routine. Suppose your Pascal program includes the following:

```
TYPE
    name = array[1..10] of char;
VAR
    first_name : name;
    len        : integer16;

procedure change_name(in out first_name : name;
                         in out        len : integer16); extern;

change_name(first_name,len);   {Assume "change_name" is a FORTRAN subroutine.}
```

This Pascal routine heading includes an "extra" parameter for the length of `first_name` that FORTRAN will add when Pascal calls the routine. The length argument must be of type **integer16** because FORTRAN's implicit length argument is an **integer*2**.

The FORTRAN routine heading does not explicitly include the length argument. For this example, it would look like this:

```
        subroutine change_name(first_name)
        character*10 first_name
```

If you send multiple strings to DOMAIN FORTRAN and you include FORTRAN's implicit length arguments in the Pascal parameter list, the length parameters must always appear at the end of the routine heading. That is, it is not correct to list them as `string1, len1, string2, len2`, etc. For instance:

```
{ Pascal program fragment. }
TYPE
    fn = array[1..10] of char;
    ln = array[1..20] of char;
         . . .
procedure process_name(in out       first_name : fn;
                          in out   middle_initial : char;
                          in out        last_name : ln;
                          in out len1, len2, len3 : integer16); extern;
         . . .
process_name(first_name, middle_initial, last_name, len1, len2, len3);
         .
         .
         .
*  FORTRAN subroutine fragment.
      subroutine process_name(first_name, middle_initial, last_name)

      character*10 first_name
      character    middle_initial
      character*20 last_name
```

7.7.4 Passing a Mixture of Data Types

The following DOMAIN Pascal program and DOMAIN FORTRAN subroutine demonstrate passing a variety of data types.

```
program pascal_fortran3;
{ This program demonstrates passing arguments of several different }
{ data types to a FORTRAN subroutine.                              }

TYPE
    last_names = array[1..10] of char;
    two_by_four = array[1..2, 1..4] of integer16;
    complex = record
                r         : real;
                imaginary : real;
              end;
    boolrec = record
                bool_var : boolean;
                a,b,c    : boolean;
              end;

VAR
    age   : integer32 := 1000000;
    lying : boolrec;
    name  : last_names := 'Tucker';
    multi : two_by_four := [ [5,8,11,14]
                             [100, 103, 106, 109] ];
    c     : complex := [4.53, 0.98];
    count1, count2, len : integer16;

procedure print_vals (in age : integer32; in lying : boolrec;
                      in name : last_names; in multi : two_by_four;
                      in c : complex);
{ This procedure prints the values of the variables. }

BEGIN

writeln ('Age = ', age:5);
writeln ('Lying = ', lying.bool_var:5);
writeln ('Name = ', name);

writeln ('Multi = ');
for count1 := 1 to 2 do
    begin
    for count2 := 1 to 4 do
        write(multi[count1,count2]:5);
    writeln;
    end; {for}

writeln ('Complex = ', c.r:4:2, ',', c.imaginary:5:2);

END;        {end procedure print_vals}

{ Note "extra" len argument in the parameter list of mixed_types.}

procedure mixed_types (in out age : integer32; in out lying : boolrec;
                       in out name : last_names; in out multi : two_by_four;
                       in out c : complex; in out len : integer16); extern;
```

```
BEGIN  {main program}
lying.bool_var := true;
writeln (chr(10), 'Before calling FORTRAN', chr(10));
print_vals(age, lying, name, multi, c);

mixed_types(age, lying, name, multi, c, len);
writeln (chr(10), 'After calling FORTRAN', chr(10));
print_vals(age, lying, name, multi, c);

END.

**********************************************************************
* This is a FORTRAN subroutine for assigning new values to arguments
* passed in from a Pascal program. It demonstrates how to pass a
* variety of data types.

      subroutine mixed_types (a,l,n,m,c)
      integer*4    a                 { Declare variables.}
      logical      l
      character*10 n
      integer*2    m(4,2), count
      complex      c

* Make reassignments.
      a = a * 2
      l = .false.
      n = 'Carter'

      do count = 1,4
          m(count,1) = m(count,1) + 1000
      enddo

      c = (2.0, -2.0)
      end
```

These programs are available on-line and are named pascal_fortran3 and mixed_types. If you compile, bind, and execute these programs, you get the following output:

```
Before calling FORTRAN

Age = 1000000
Lying =  TRUE
Name = Tucker
Multi =
    5    8   11   14
  100  103  106  109
Complex = 4.53, 0.98

After calling FORTRAN

Age = 2000000
Lying = FALSE
Name = Carter
Multi =
 1005 1008 1011 1014
  100  103  106  109
Complex = 2.00,-2.00
```

7.8 Calling a C Routine from Pascal

In addition to allowing you to call FORTRAN routines, DOMAIN Pascal permits you to call routines written in DOMAIN C source code. To accomplish this, perform the following steps:

1. Write source code in DOMAIN Pascal that calls a routine. Compile it with the DOMAIN Pascal compiler. DOMAIN Pascal creates an object file.
2. Write source code in DOMAIN C. Compile it with the DOMAIN C compiler. DOMAIN C creates an object file.
3. Bind the object file(s) the Pascal compiler created with the object file(s) the C compiler created. The binder creates one executable object file.
4. Execute the object file as you would execute any other object file.

The remainder of this chapter describes steps 1 and 2. For information on steps 3 and 4, see Chapter 6.

NOTE: The following sections explain how to call DOMAIN C from DOMAIN Pascal. If you want to learn how to call Pascal from C, see the *DOMAIN C Language Reference*.

7.8.1 Reconciling Differences in Argument Passing

Pascal usually passes arguments by reference, while C usually passes them by value. In order to pass arguments by reference correctly, you must declare your parameters in C to be pointers so that they can take the addresses Pascal passes in. The examples in the following sections demonstrate how to do this.

If you want to pass your Pascal arguments by value, you must use the **val_param** routine option in your procedure or function heading. Chapter 5 describes **val_param**.

7.8.2 Case Sensitivity Issues

When the DOMAIN Pascal compiler parses a program, it makes all identifier names uppercase, regardless of the way you type the names in your source code. In contrast, when the DOMAIN C compiler parses a program, it *inverts* the case of all identifiers. So if your C source code includes the following

```
double square_num(a,b)
       float *a,*b;
```

the compiler converts the function name `square_num` and the variables `a` and `b` to `SQUARE_NUM`, `A`, and `B`. But if the function heading looks like this

```
double SQUARE_NUM(A,B)
       float *A,*B;
```

the compiler converts `SQUARE_NUM`, `A`, and `B` to `square_num`, `a`, and `b`. It is important to understand this when you are calling C from Pascal. In order to make identifier names match up at bind time, you should *always* use lowercase letters in your C subprograms. That way, the C compiler will invert them to uppercase and they will match the always-uppercased identifier names in the Pascal program.

7.9 Data Type Correspondence for Pascal and C

Before you try to pass data between DOMAIN Pascal and DOMAIN C, you must understand how Pascal data types correspond to C data types. Table 7-2 lists these correspondences.

Table 7-2. DOMAIN Pascal and DOMAIN C Data Types

DOMAIN Pascal	DOMAIN C
char	**char**
integer, integer16	**short**
integer32	**int, long**
real, single	**float**
double	**double**
enumerated types	**enum**
record	**struct**
variant record	**union**
pointer(^)	pointer(*)
boolean	none
set	none
none	**unsigned char**
0..65335	**unsigned short**
0..4295967295	**unsigned long**

As the table shows, the integer, real, and character data types in both languages correspond very well. For example, Pascal's **integer16** data type is identical to C's **short** data type, and a **double** variable in Pascal is the same as a **double** in C.

However, there are some important differences. DOMAIN C has no equivalent types for Pascal's **boolean** or **set** types, although you can simulate these types.

There also is a C type that has no Pascal equivalent. There is no easy way to simulate C's **unsigned char** type in Pascal. Therefore, if you pass an **unsigned char** value to a Pascal program, it is interpreted as a signed value. This only makes a difference when the high-order bit is set.

7.9.1 Passing Integers and Real Numbers

Since the Pascal and C integer and real data types match up so well, it is fairly easy to pass data of these types between the two languages. Make sure that all arguments agree in type and size, either by declaration or by casting.

The example below shows a Pascal program that solicits the values for two sides of a right triangle. It then sends those values into the C function, which computes and returns the length of the hypotenuse. The arguments for the triangle's sides are 32 bits each, while the result is 64 bits.

```
PROGRAM pascal_c1;
{ This program calls a C function that calculates the hypotenuse of }
{ a right triangle when a user supplies the lengths of two sides.   }

VAR
    leg1, leg2 : single;

function hypot_c(in out leg1, leg2 : single) : double; extern;

BEGIN

writeln ('This program calculates the hypotenuse of a right triangle ');
writeln ('given the length of its two sides.');
write ('Enter the lengths of the two sides: ');
readln (leg1,leg2);

writeln ('The triangle''s hypotenuse is: ', hypot_c(leg1,leg2):5:2);

END.


/*************************************************************/
/* This is a C function for finding the hypotenuse of a      */
/* right triangle. The arguments must be declared as pointers.*/
#include <math.h>
double hypot_c(a,b)
       float *a,*b;
{
double result;
result = sqrt((*a * *a) + (*b * *b));
return(result);
}
```

These programs are available on-line and are named pascal_c1 and hypot_c. Following is a sample execution of the bound program.

```
This program calculates the hypotenuse of a right triangle
given the length of its two sides.
Enter the lengths of the two sides: 3 4
The triangle's hypotenuse is:  5.00
```

7.9.2 Passing Character Arguments

Passing arguments when two languages' data types match exactly is relatively easy, but passing them when they don't often means you have to do extra work.

You must do extra work when you pass character strings between the languages. C automatically appends a null character, \0, to the end of every string, so if your Pascal program is going to print a string it receives from C, you must remember to take care of the null. You can strip it off either in your C routine or in Pascal; the sample program in this section shows how to strip it off in C.

The following Pascal program sends two strings to a C routine that prompts a user for new values and then sends the new string values back.

```
PROGRAM pascal_c2;
{ This program demonstrates passing character variables to and }
{ getting them back from a C routine.                          }

TYPE
    fn = array[1..10] of char;
    ln = array[1..15] of char;
VAR
    first_name : fn;
    last_name  : ln;

procedure pass_char(in out first_name : fn;
                    in out last_name  : ln); extern;

BEGIN
first_name := 'Sherlock';
last_name := 'Holmes';

writeln ('Before calling C, this is the name: ', first_name, last_name);
pass_char(first_name, last_name);
writeln ('After calling C, this is the name: ', first_name, last_name);

END.


/***********************************************************************/
/* This C routines takes two strings, prompts a user for new values,  */
/* and strips off the null characters before sending the strings back.*/
#include <stdio.h>
pass_char(first_name, last_name)
    char *first_name, *last_name;

{
short i, j;
char hold_first[10], hold_last[15];

printf ("\nEnter the first name and last name of a detective: ");
scanf ("%s%s", hold_first, hold_last);

/* Strip off the null character C automatically appends to any string */
/* and blank out any previously used places in the name strings.      */
for (i = 0; hold_first[i] != '\0'; i++)
    first_name[i] = hold_first[i];
for (j = i; j < 10; j++)
    first_name[j] = ' ';

for (i = 0; hold_last[i] != '\0'; i++)
    last_name[i] = hold_last[i];
for (j = i; j < 15; j++)
    last_name[j] = ' ';
}
```

These programs are available on-line and are named pascal_c2 and pass_char. Following is a sample execution of the bound program.

```
Before calling C, this is the name: Sherlock  Holmes

Enter the first name and last name of a detective: Philip Marlowe
After calling C, this is the name: Philip    Marlowe
```

7.9.3 Passing Arrays

Single-dimensional arrays (except for **boolean** arrays) of the two languages correspond fairly well. The major difference is that in C, array subscripts always begin at zero, while Pascal allows the programmer to determine the subscript at which the array begins. In order to make arrays match up, you should define your Pascal subscripts to begin at zero. For example:

In DOMAIN Pascal	*In DOMAIN C*
`x = array[0..9] of char`	`char x[10]`
`x = array[0..49] of integer`	`short x[50]`
`x = array[0..19] of single`	`float x[20]`

With such declarations, the following code fragments access the identical elements in an array:

In DOMAIN Pascal

```
for i := 0 to 9 do
    my_array[i] := i;
```

In DOMAIN C

```
for (i = 0; i < 10; i++)
    my_array[i] = i;
```

As described earlier, Pascal by default passes arguments by reference, so when it sends an array argument to a routine, it actually is sending the address of the first element in the array. C gets that address when you declare the array variable in C to be a pointer. This means you don't have to specify the size of a single-dimensional array that your C subprogram receives from Pascal. That is, if your Pascal program includes the following

```
type
    x = array[0..9] of integer32;
var
    my_array : x;
        . . .
pass_array(my_array);
```

your C routine heading can look like this:

```
pass_array(my_array)
    long *my_array;        /*Notice that there's no indication of the
                             array's dimensions.                      */
```

The following example shows a Pascal program that loads five user–entered scores into a single–dimensional array and then sends that array to a C procedure to compute the average. Notice that the argument size determines the dimension of the array; the C declaration of the array contains no dimensioning information.

```
PROGRAM pascal_c3;
{ This program demonstrates passing a single-dimensional }
{ array to a C routine.                                  }

TYPE
    scores = array[0..4] of integer;
VAR
    grades : scores;
    i, j   : integer;
    result : single;
    size   : integer := 5;

procedure single_dim(out result : single;
                     in    size : integer;
                     in  grades : scores); extern;

BEGIN
writeln ('Enter ', size:1, ' integer test scores separated by spaces.');
for i := 0 to 4 do
    read(grades[i]);
readln;

single_dim(result, size, grades);
writeln ('C computed the average of the test scores, and it is: ',
         result:5:2);

END.

/*********************************************************************/
single_dim(result,size,grades)
    float *result;
    short *size, *grades;
{
short i,total;
total = 0;
for (i = 0; i < *size; i++)      /* Add up array values */
    total += grades[i];          /* and then compute    */
*result = total / 5.0;           /* average.            */
}
```

These programs are available on–line and are named pascal_c3 and single_dim. Following is a sample execution of the bound program.

```
Enter 5 integer test scores separated by spaces.
85 92 100 79 96
C computed the average of the test scores, and it is: 90.40
```

Multidimensional arrays in the two languages also correspond fairly well. Both languages store such arrays in the same order; that is, the right–most subscript varies fastest. So these two arrays would be stored identically:

In DOMAIN Pascal	*In DOMAIN C*
`x = array[0..1,0..2] of integer32;`	`long *x[2][3];`

7.9.4 Passing Pointers

Passing pointers between Pascal and C is fairly straightforward. In both cases, pointers are 4–byte entities. The following example shows a simple linked–list application. The Pascal program creates the first element of the list and then calls the C routine append to add new elements to the list. The routine `printlist` is a Pascal routine that prints the entire list. In addition to illustrating how to pass pointers, this example also shows the correspondence of Pascal records to C structures.

```
PROGRAM pascal_c4;
TYPE
    link = ^list;
    list = record
           data : char;
           p : link;
           end;

VAR
    last_let     : char := 'z';
    val          : char;
    base, first  : link;

procedure append(in     base : link;
                 in      val : char); extern;


procedure printlist;
{ printlist prints the data in each member of the linked list. }

BEGIN

while base <> nil do
    begin
    writeln(base^.data);
    base := base^.p;
    end;

END;        {end procedure printlist}


BEGIN       {main program}
base := nil;
new(first);

first^.data := 'a';           { Assign value to first element of the list. }
first^.p := base;             { The first element is also the last, so set }
                              { pointer to nil.                            }
base := first;                { Base points to the beginning of the list.  }

val := 'b';
append(base,val);
append(base,last_let);

printlist;                    { Call procedure to print contents of list.  }

END.
```

```
/**********************************************************************/
/* C routine that appends items to a linked list.                    */

#module pass_pointers_c
#include <stdio.h>

typedef struct
    {
    char data;
    struct list *next;
    } list;

void append (base, val)
    list **base;
    char *val;

{
list *newdata, *last_rec;

last_rec = *base;                        /* Point temp variable last_rec at  */
                                         /* the beginning of the list.       */
newdata = (list*)malloc(sizeof(list));  /* Allocate memory for new element. */

while (last_rec->next != NULL)           /* Walk to the end of the list.     */
    last_rec = last_rec->next;

last_rec->next = newdata;                /* Add new data.                    */
newdata->data = *val;
newdata->next = NULL;

}
```

These programs are available on-line and are named pascal_c4 and append. If you compile and bind the programs, and execute the result, you get this output:

```
a
b
z
```

Chapter 8

Input and Output

You may have turned to this chapter first if you are new to DOMAIN Pascal yet experienced with other Pascals. After all, I/O is the most system-dependent aspect of any Pascal. I/O on DOMAIN Pascal may only partially resemble I/O on some other Pascal. These are necessary differences because each implementation of Pascal must take advantage of the features of the host operating system.

DOMAIN Pascal supports the following three methods of performing I/O:

- Input Output Stream (IOS) calls
- VFMT calls
- Predeclared DOMAIN Pascal I/O procedures

In general, you can perform all your I/O with the predeclared DOMAIN Pascal I/O procedures. However, the other two methods can be very useful in certain circumstances.

This chapter provides a brief overview of all three methods, along with some background information that may aid you in whatever method you choose.

8.1 Some Background on DOMAIN I/O

This section describes some information that may be helpful in understanding how I/O works on the DOMAIN system. It's only a brief sketch; the full details are published in *Programming With General System Calls*. Before describing the mechanics of DOMAIN I/O here, there is a brief description of IOS calls and VFMT calls.

8.1.1 Input Output Stream (IOS)Calls

IOS calls are system calls that perform I/O. You can easily make IOS calls from your DOMAIN Pascal program. IOS calls can:

- Create a file
- Open or close a file
- Write to or read from a file
- Change a file's attributes (a file's attributes include name, length, type uid, accessibility, etc.)
- Access magnetic tape files or serial lines

IOS calls are more primitive than the predeclared DOMAIN Pascal I/O procedures. Consequently, they give you more control over I/O, but they are harder to use. Therefore, for simple I/O needs you are probably better off using the predeclared DOMAIN Pascal I/O procedures. If you want to do something out of the ordinary, then you will most likely need to use IOS calls.

Chapter 3 of *Programming With General System Calls* details IOS.

8.1.2 VFMT Calls

VFMT calls are special system calls that format input and output. Since the Pascal language does not support elaborate formatting features, you may find it useful to make a VFMT call in situations such as the following:

- A variable contains a hexadecimal value and you wish to prompt your user with its ASCII equivalent.
- You want to tabulate results in fixed columns using scientific notation.
- You need to parse an input line without worrying about whether the user separates the arguments with spaces or semicolons.

VFMT is a set of tools for converting data representations between formats.

VFMT performs two classes of operations -- encoding and decoding. Encoding means taking program-defined variables and producing text strings that represent the values of the variables, in a format that you specify. These encoded values are then often written to output for viewing. Decoding means taking text (typically typed by the user), interpreting it in a way that you specify, and storing the apparent data values in program-defined variables.

The VFMT calls allow you to format the following kinds of data:

- ASCII characters
- 2-byte or 4-byte integers interpreted as signed or unsigned integers in octal, decimal, or hexadecimal bases
- single- and double-precision reals in floating-point and scientific notations

This includes the following DOMAIN Pascal data types: **char**, **integer16**, **integer32**, **single**, and **double**.

Chapter 8 of *Programming With General System Calls* details VFMT calls. In addition, you can use **getpas** (see Chapter 1) to obtain copies of several sample programs that contain VFMT calls.

The remainder of this section is devoted to explaining certain aspects of DOMAIN I/O that you may find useful.

8.1.3 File Variables and Stream IDs

All of the predeclared DOMAIN Pascal I/O procedures take a file variable as an argument. The file variable is a synonym for a temporary or permanent pathname to the file. If you are using IOS calls rather than the DOMAIN Pascal I/O procedures, you refer to a pathname by its IOS ID rather than by its file variable. A stream ID is a number assigned by the operating system when you open a file or device. Since DOMAIN Pascal I/O procedures in your source code ultimately translate to IOS calls at runtime, a file variable in your source code becomes a IOS ID at runtime. The system can support up to 128 I/O streams per process.

8.1.4 Default Input Output Streams

Every process starts out with the I/O streams shown in Table 8-1. DOMAIN Pascal deals in file variables, not IOS IDs, so the table also shows the names of the file variables corresponding to these streams. You need not explicitly declare these; the system opens these streams automatically as described in the next subsection.

Table 8-1. The Default Streams

Stream Name	File Variable Name	Description
ios_$stdin	**Input**	If you do not explicitly specify a file variable for a DOMAIN Pascal input procedure, the compiler reads from **input**. By default, **input** is the process input pad, but you can redirect this stream with the < character.*
ios_$stdout	**Output**	If you do not explicitly specify a file variable for a DOMAIN Pascal output procedure, the compiler assumes **output**. By default, the system associates this stream with the process transcript pad, but you can redirect this stream with the > character.*
ios_$errin		This is just another input stream available for you to use. (It has nothing to do with errors.) By default, **errin** is the process input pad, but you can redirect this stream with the <? character sequence.*
ios_$errout		DOMAIN Pascal sends errors to this stream. By default, this is the transcript pad, but you can redirect this stream with the >? character sequence.*

8.1.5 Interactive I/O

DOMAIN Pascal uses the following system for interactive processing of the standard input (**input**) and standard output (**output**) files. DOMAIN Pascal does not actually open **input** and **output** until the program first refers to them. When DOMAIN Pascal finds the first reference to **input** in your program, it

* The *DOMAIN System User's Guide* explains how to redirect I/O.

calls **reset(input)**. **Reset** expects to fill the file buffer variable with the first **char** of a text file, which means that **reset(input)** can cause a request for input. For example, consider the following program:

```
PROGRAM lazy;                  {This program is WRONG!}
VAR
    c : char;
BEGIN
while not eof(input) do
    begin
    write('enter a letter or an EOF --');
    readln(c);
    end;
END.
```

If you run this program, you might expect results like the following:

```
enter a letter or an EOF -- A
enter a letter or an EOF -- Z
enter a letter of an EOF -- <EOF>
```

However, in reality, the program does not produce those expected results. That's because the first reference to **input** is in the **eof** function. This causes the system to perform a **reset(input)** prior to the test for **eof**. **Reset** expects to fill the file buffer, so the test for **eof** actually results in a request for input. Therefore, running the program results in the following:

```
A
enter a letter or an EOF -- Z
enter a letter or an EOF -- <EOF>
```

To eliminate this problem, you must take advantage of a feature of **reset** known as delayed access. Delayed access means that data will not be supplied to fill the input buffer at the **reset**, but rather at the next reference to the file. Since **reset** initiates delayed access, and since **eof** and **eoln** cause the file buffer to be filled, you must place the first prompt for input *before* any tests for **eof** or **eoln**. The data you enter in response to the prompt is retained until you make another reference to the input file.

The following shows how to use delayed access to make the previous example work correctly:

```
PROGRAM interactive;
VAR
    c : char;
BEGIN
write ('enter a letter or an EOF -- ');
while not eof(input) do
    begin
    readln(c);
    write ('enter a letter or an EOF -- ');
    end;
END.
```

8.1.6 Stream Markers

When you open a file, the operating system assigns a stream marker to the file. A stream marker is a pointer that points to the current position inside the open file. As you read from or write to the file, the operating system moves the stream marker forward in the file. The stream marker points to the byte (or record) that the system can next access. When you open the file with the DOMAIN Pascal **open** procedure, the stream marker initially points to the beginning of the file. Using IOS calls, you can open the file with the stream marker initially pointing to the end of file so that you can append to the file.

If you are using IOS calls, you directly control the stream marker. If you are using DOMAIN Pascal I/O procedures, you control the stream marker indirectly through the procedures you call.

8.1.7 File Organization

The DOMAIN operating system supports the following four types of file organization:

- UASC context delimited ASCII record files ("UASC file" for short)
- Fixed-length records files ("rec file" for short)
- Variable-length records files
- No defined record structure files

Using stream calls, you can create any of the four types of files. However, using DOMAIN Pascal, you can create only the first two types.

A UASC file is an ordinary text file. The system stores a text file as a 32-byte header followed by ASCII characters. The operating system makes no attempt to organize or structure the data in a text file. That is, '908' is stored in the three bytes it takes to hold the ASCII values of digit '9', digit '0', and digit '8', rather than structuring it into the value of integer 908.

It is a DOMAIN Pascal restriction that each line of a text file be no longer than 256 characters. By line, we mean all the characters between two end-of-line characters. No text file can have more than 32767 lines.

A rec file consists of zero or more "records." The term record is confusing here. In this context it does not necessarily mean a Pascal record variable. Instead, it means a collection of some data type. When you create a rec file with DOMAIN Pascal I/O procedures, the data type of the record is the same as the base type of the file variable. In a fixed-length record file, each record must be of the same type, and must be of the same size.

8.2 Predeclared DOMAIN Pascal I/O Procedures

The encyclopedia of DOMAIN Pascal code in Chapter 4 details the syntax of each of the predeclared DOMAIN Pascal I/O procedures. This section tries to provide a global view of these procedures.

8.2.1 Creating and Opening a New File

You can create a permanent file or a temporary file. The operating system deletes a temporary file as soon as the program that created it ends. Permanent files last beyond program execution. In fact, they last until you explicitly delete them.

To create a permanent file, you call the **open** procedure and specify 'NEW' as the file_history. This not only creates the file, but opens it for future access as well.

To create a temporary file, you call the **rewrite** procedure.

Both **open** and **rewrite** take a **file** or **text** variable as an argument. If the file variable has the **text** data type, then DOMAIN Pascal creates an UASC file. If the file variable has the **file** data type, DOMAIN Pascal creates a rec file.

8.2.2 Opening an Existing File

To open an existing file for future access, you call the **open** procedure and specify either 'OLD' or 'UNKNOWN' as the file_history.

Note that you do not have to explicitly open the Shell transcript pad. It is already open. (See the "Default Input Output Streams" section earlier in this chapter for details.)

8.2.3 Reading From a File

In order to read from an open file, you must call the **reset** procedure. **Reset** tells the system to treat the open file as a read-only file (with the one exception being the **replace** procedure). You can change the open file to a write-only file with the **rewrite** procedure.

After calling **reset**, you are free to call any or all of the three input procedures that DOMAIN Pascal supports, namely, **read**, **readln**, and **get**. All three procedures read information from the specified file and assign it the specified variable(s). The following list describes the differences among the three procedures:

- **Get** can access both rec files and UASC files. Use it to assign the contents of the next record or character in a file to a file buffer variable.
- **Read** can access both rec files and UASC files. Use it to read information from the specified file into the given variables. After reading a record (if a rec file) or character (if a UASC file), **read** positions the stream marker to point to the next record or character in the file.
- **Readln** can access UASC files only. It is similar to **read** except that after reading the information, **readln** sets the stream marker to the character immediately after the next end-of-line character.

It is often useful to know when the stream marker has reached the end of the line or the end of the file. You can use **eoln** to test for the end of line, and **eof** to test for the end of file in UASC files.

DOMAIN Pascal supports the **find** procedure as an extension to standard Pascal. Use it to set the stream marker to point to a particular record in a rec file. This procedure permits you to skip randomly through a rec file, while the other read procedures imply a sequential path.

8.2.4 Writing to a File

In order to write to a file, you must call the **rewrite** procedure. (If you used **rewrite** to open a temporary file, then you don't have to call **rewrite** again.) The **rewrite** procedure tells the system to treat the open file as a write-only file. You cannot read from this file unless you call **reset**.

Once the file has been opened for writing, you can call any of these four standard Pascal output procedures: **write**, **writeln**, **put**, and **page**. **Write**, **writeln**, and **put** are the output mirrors to **read**, **readln**, and **get**. Using **write**, **writeln**, or **put** causes DOMAIN Pascal to write the specified information from the specified variable(s) to the specified file. Here are the differences among the four procedures:

- **Put** can access both rec files or UASC files. Use it to assign the contents of the file buffer variable to the next file position, causing the contents to be written to the file.
- **Write** can access both rec files and UASC files. Use it to write information from the specified variables into the specified file. After writing a record (if a rec file) or character (if a UASC file), **write** positions the stream marker to point to the next record or character in the file.
- **Writeln** can access UASC files only. It is similar to **write** except that after writing the information, **writeln** sets the stream marker to the character immediately after the next end-of-line character.
- **Page** can access UASC files only. Use it to insert a formfeed (page advance) into the file.

In addition to the standard Pascal output procedures, DOMAIN Pascal also supports the **replace** procedure. Use the **replace** procedure to substitute a new record for an existing record in a rec file. **replace** has the distinction of being an output procedure that you can call only when the file has been open for input. In other words, before you call **replace**, you must first call **open** and **reset** to open the file for reading. **Replace** is usually used with **find**. Use **find** to skip through a rec file looking for a particular record, then use **replace** to modify the record in its place.

8.2.5 Closing a File

When a program terminates (naturally or as a result of a fatal error), the operating system automatically "closes" all open files. "Closing" means that the operating system unlocks the file. When the operating system closes a rec file, it automatically preserves any changes made to the rec file. However, when the operating system closes a text file that was open for output, there is a possibility that some modifications won't be preserved. To ensure that all modifications are kept, make sure that the last output operation on the file is a **writeln**.

DOMAIN Pascal supports a **close** procedure whose purpose is to close a specified open file. Since the operating system does this automatically at the end of the program, you ordinarily don't have to call **close**. However, it is good programming practice to close all open files as soon as your program is finished using them. Open files tie up process resources and may cause your program to needlessly lock a file that another program wants to access.

Chapter 9

Errors

The majority of this chapter is devoted to detailing compiler errors and warnings. However, we start this chapter with a discussion of the errors reported by the predeclared procedures **open** and **find**.

9.1 Errors Reported by Open and Find

The **open** and **find** procedures are the only two predeclared DOMAIN Pascal routines that return an error status parameter. This parameter tells you whether or not the call was successful. If the call was successful, the operating system returns a value of 0 in the error status parameter. If the call was not successful, the operating system returns a number symbolizing the error. Your program is responsible for handling the error. You may wish to print the error and terminate execution. Possibly, you may wish to code your program so that it can take appropriate action when it encounters an error.

This error status parameter is identical to the error status parameter returned by all system calls. This is more than coincidental since **open** and **find** are executed as stream calls at runtime. For complete details on using the error status parameter, refer to *Programming With General System Calls*. For an overview relevant to **open** and **find**, read the following section.

9.1.1 Printing Error Messages

To print an error message generated by an errant **open** or **find**, you must do the following:

- Put the following two include directives in your program just after the program heading:

  ```
  %INCLUDE '/sys/ins/base.ins.pas';
  %INCLUDE '/sys/ins/error.ins.pas';
  ```

  Make sure you put `base.ins.pas` before `error.ins.pas`.

- Declare the error status parameter with the `status_$t` data type; for example,

```
VAR
   err_stat : status_$t;     {status_$t is declared in base.ins.pas}
```

- Specify the error status parameter as the `all` field of the error status variable; for example,

```
OPEN(f, pathname1, 'NEW', err_stat.all);
```

- Call the `error_$print` procedure with the error status parameter as its sole argument; for example,

```
error_$print(err_stat);
{The error_$print procedure is defined in error.ins.pas}
```

The following program puts all the steps together:

```
Program test;

%INCLUDE '/SYS/INS/BASE.INS.PAS';
%INCLUDE '/SYS/INS/ERROR.INS.PAS';

VAR
          f : text;
   err_stat : status_$t;

BEGIN
    OPEN(f, 'grok', 'OLD', err_stst.all);
    Error_$print(err_stat);
END.
```

The `error_$print` procedure writes the error or warning message to **stdout**. If there is no error or warning, `error_$print` writes the following message to **stdout**:

```
status 0 (OS)
```

9.1.2 Testing For Specific Errors

The previous subsection introduced the `status_$t` data type and its `all` field. This section describes another field in `status_$t` -- the code field. The code field of `status_$t` contains a number that corresponds to a particular error. To test for a specific error, compare this code field against expected errors. Table 9-1 lists the common error codes returned by **open** and Table 9-2 lists the common error codes returned by **find**. The symbolic names come from the */sys/ins/streams.ins.pas* file. To use these symbolic names, all you have to do is list this file as an include file.

Table 9-1. Common Error Codes Returned By Open

Code	Symbolic Name	Cause of Error
1	**stream_$not_open**	You specified a file_history of 'NEW', but the pathname existed. Furthermore, the type UID (rec file or USAC) of this existing file differed from the type UID of the file you were trying to open.
14	**stream_$already_exists**	You specified a file_history of 'NEW', but the pathname already exists.
21	**stream_$name_not_found**	You specified a file_history of 'OLD', but the pathname does not exist.
28	**stream_$object_not_found**	You specified a file_history of 'OLD', but the operating system cannot locate the disk containing the pathname. (Indicates network problems.)
45	**stream_$insufficient_rights**	The ACL of the pathname prohibits you from opening the file.

Table 9-2. Common Error Codes Returned By Find

Code	Symbolic Name	Cause of Error
1	**stream_$not_open**	You called **find**, but without the file being open.
9	**stream_$end_of_file**	You specified a record number greater than the number of records in the file.

For example, consider the following program fragment, which tries to first open pathname my_book1. If this pathname exists, the program then attempts to open pathname my_book2.

```
Program test;

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';

VAR
     f1, f2 : text;
     st     : status_$t;

BEGIN
   open(f1, 'my_book1', 'NEW', st.all);
   if st.code = stream_$already_exists then
       open(f2, 'my_book2', 'NEW', st.all);
   . . .
```

9.2 Compiler Errors and Warnings

When you compile a program, the compiler reports errors and warnings.

An error indicates a problem severe enough to prevent the compiler from creating an executable object file.

A warning is less severe than an error; a warning does not prevent the compiler from creating an executable object file. The warning message tells you about a possible ambiguity in your program for which the compiler believes it can generate the correct code.

The following pages list the common DOMAIN Pascal compiler error and warning messages, and suggest ways to handle them. In addition, remember the cardinal rule of Pascal debugging:

LOOK FOR MISSING SEMICOLONS

For example, suppose the compiler reports an error at line 50. Further, suppose that you stare at line 50 in disbelief, knowing full well that you have written a perfect statement. In frustration, you pick up this manual and throw it out the window. A friend comes in, looks at you, looks at your program, and says "You forgot a semicolon at line 49." The day is saved, you get the promotion, and you send the technical writer a note of apology.

9.2.1 Error and Warning Message Conventions

The error and warning messages listed in the rest of this chapter follow these conventions:

- Keywords in the message text are capitalized, since that's the way they appear on your screen. In the accompanying explanatory text, they are lowercase bold, as they are elsewhere in this manual.

- Italicized words in the message text indicate values that the compiler fills in when generating the message. For example, suppose your program contains the following:

```
PROGRAM err_test;
VAR
    num  :  integer17;
  .  .  .
```

  Because the fragment includes an undefined data type (integer17), it triggers Error 23, which reads:

  23   ERROR   *Identifier* has not been declared in routine *name_of_routine*.

  When you compile, *identifier* and *name_of_routine* are filled in like this:

```
(0003)     num  :  integer17;
******** Line 3: [Error 023]  INTEGER17 has not been declared in
         routine ERR_TEST.
```

9.2.2 Error and Warning Messages

Following are DOMAIN Pascal's compiler error and warning messages.

1	ERROR	Unterminated comment.
		You started a comment, but you did not close it, or you closed it with the wrong delimiter. Comment delimiters must match unless you compile with the **-iso** switch. If you don't compile with that switch and you start a comment with {, you must end it with }. Similarly, if you start a comment with (*, you must end it with *). If you start a comment with ", you must end it with ".
2	ERROR	Improper numeric constant.
		You specified a base that fell outside the legal range of 2 to 16. For example, you cannot specify a number in base 32. Perhaps you mistakenly specified the integer first and the base second. (See the "Integers" section in Chapter 2 for an explanation of base.)
3	ERROR	Unterminated character string.
		You started a string with an apostrophe ('), but you forgot to end it with an apostrophe. See Chapter 2 for a definition of string.
4	ERROR	Bad syntax *(token)*.
		The compiler encountered *token* when it was expecting to find something else.
6	ERROR	Period expected at end of program *(symbol)*.
		You must finish the program with an **end** statement followed by a period. *symbol* is the final character that the compiler found in your source code. Typically, *symbol* is /EOF/ (an end-of-file character) or a semicolon. The most frequent cause of this error is putting a semicolon rather than a period after the final **end**.
7	ERROR	Text following end of program *(token)*.
		You have put some text other than a comment after the end of the program. The phrase "END." marks the end of the program. You can put comments after the end of the program, but you cannot put anything else there.
8	ERROR	PROGRAM or MODULE statement expected *(token)*.
		The first noncomment in your source code must be either a **program** heading or a **module** heading. The compiler found *token* instead of a **program** or **module** heading. DOMAIN Pascal also issues this error when there is some sort of mistake in your **program** or **module** heading. Chapter 2 describes the **program** heading. Chapter 7 describes the **module** heading.
10	ERROR	Semicolon expected at end of **program/module** statement *(token)*.
		You forgot to end your **program** or **module** heading with a semicolon. DOMAIN Pascal encountered *token* when it was expecting a semicolon.

11	ERROR	Improper declarations syntax *(token)*.

DOMAIN Pascal found an unexpected *token* when processing a declaration part.

12	ERROR	Improper CONST statement syntax *(token)*.

You made a mistake when declaring a constant. See Chapter 2 for the correct format. *Token* is the invalid token that DOMAIN Pascal encountered. A possible trigger for this error is that you tried to declare two identifiers for the same constant value as in the following example:

```
CONST
   x,y = 5;
```

13	ERROR	Improper TYPE statement syntax *(token)*.

You made a mistake in the **type** declaration part of your program. DOMAIN Pascal encountered *token* when it was expecting something else. See Chapter 2 for the correct format of the **type** declaration part. A possible trigger for this error is that you used the wrong symbol to associate the identifier with its type as in the following example:

```
TYPE
   long := integer32;
```

14	WARNING	*Datatype* cannot be PACKED.

You used the keyword **packed** in front of some data type other than **record**. The only data type DOMAIN Pascal packs is **record**. *Datatype* is the data type you tried to pack. In order to conform to the ISO standard, DOMAIN Pascal recognizes the **packed** syntax for data types other than **record**, but it does not actually pack those other data types.

15	ERROR	Improper type specification *token*.

You made some mistake when specifying a data type in the **var** declaration part. *Token* identifies the unexpected part of the **var** declaration part. For example, the following declaration triggers this error because r is a variable, not a data type:

```
VAR
    r : real;
    data : array[1..10] of r;
```

See Chapter 2 for a complete description of the **var** declaration part.

16	ERROR	Improper enumerated constant syntax *(message)*.

All constants in an enumerated type must be valid identifiers. See Chapter 2 for a definition of identifier. *Message* identifies the first character or token that did not conform to the rules for identifiers.

17	ERROR	OF expected in SET specification *(token)*.

When you declared a set type or set variable, you forgot to specify the keyword **of** in between the word **set** and the base type. Refer to Chapter 3 for information on declaring set types. DOMAIN Pascal encountered *token* rather than the keyword **of**.

18	ERROR	Improper ARRAY specification syntax *(token)*.

You declared an array incorrectly. See Chapter 3 for details on declaring arrays. *TOKEN* is the token that DOMAIN Pascal encountered when it expected to find something else.

19	ERROR	Improper RECORD specification syntax *(token)*.

You declared a record incorrectly. See Chapter 3 for details on declaring records. DOMAIN Pascal encountered *token* when it expected to find something else. A common trigger of this error is a type declaration such as the following:

```
TYPE
    student = record
      a : integer32;
      b = boolean;{cause of error. '=' should be ':'}
   end;
```

20	ERROR	Improper pointer specification *(token)*.

You declared a pointer incorrectly. See Chapter 3 for details on pointing records. DOMAIN Pascal encountered *token* when it expected to find something else. The following type declaration triggers this error, because the uparrow (^) can only appear before a type name, not a type specification:

```
TYPE
    ptr_to_ptr_to_long = ^^integer32;
```

21	ERROR	Improper VAR statement syntax *(token)*.

In a variable declaration, you probably forgot to specify a semicolon after the data type. Perhaps you specified a comma instead of a semicolon, or perhaps you did not specify any punctuation mark at all. This error also occurs if you begin an identifier with a digit (0–9) or dollar sign ($).

22	ERROR	Parameter list must only be specified when the procedure or function is declared as FORWARD.

You specified the parameter list for this routine in two places: first when you specified **forward** or **extern** and second in the routine heading itself. To correct this error, eliminate the second parameter list.

23 ERROR *Identifier* has not been declared in routine *name_of_routine*.

Several conditions can trigger this error. You might have used *identifier* in the code portion of *name_of_routine* without *identifier* being accessible to this routine. Study the "Global and Local Variables" and "Nested Routines" sections in Chapter 2 to learn about the scope of declared identifiers.

Another possible trigger for this error is that you tried to make a forward call to a procedure without declaring the procedure with the **forward** attribute. (See the "Routine Attribute List" section in Chapter 5 for details on the **forward** attribute.)

This error also can occur if you specify a data type that either is invalid or that you've forgotten to define in the **type** section of your program.

24 ERROR Multiple declaration of *identifier*, previous declaration was on line *line_number*.

You declared *identifier* (which could be a data type, variable, constant, label, or routine) more than once.

25 ERROR Improper MODULE structure *(token)*.

The compiler was expecting to encounter a procedure or function declaration, but found *token* instead. Remember, that unlike a program, the action part of a module must always be contained inside a named routine.

26 ERROR Number of array subscripts exceeds limit of seven.

DOMAIN Pascal supports arrays of up to seven dimensions.

29 ERROR Subrange bound *(token)* is not scalar.

You were trying to declare an array or subrange, but one of the bounds of the subrange was not a scalar. The scalar types are integer, Boolean, char, enumerated, and subrange. The *token* is the token that DOMAIN Pascal encountered when it was searching for a scalar expression.

30 ERROR Lower bound of subrange *(lower_bound)* is not of the same type as the upper bound *(upper_bound)*.

You were trying to declare an array or subrange, but the data type of *lower_bound* is not the same data type as that of *upper_bound*. The types must match.

31 ERROR Lower bound of subrange *(left_scalar)* is greater than the upper bound *(right_scalar)*.

You were trying to declare an array or subrange, but you set the value of the *left_scalar* to a higher value than the value of the *right_scalar*. The *left_scalar* must be lower than the *right_scalar*.

32 ERROR Base type *(type)* of SET is not scalar.

You tried to declare a nonscalar type as the base type of a set variable or type. The scalar types are integer, char, Boolean, enumerated, and subrange. *Type* is the token that DOMAIN Pascal encountered instead of a scalar type.

33 ERROR SET elements must be positive (–).

You tried to declare a set with a base type of integer or subrange, but DOMAIN Pascal discovered a negative number in the base type.

34 ERROR SET exceeds limit of 256 elements (*(token)*).

DOMAIN Pascal cannot store a set that exceeds 256 elements. See the "Internal Representation of Sets" section in Chapter 3 for details.

35 ERROR Improper use of *(identifier)*, only a TYPE defined name is valid here.

The compiler was expecting a data type, but you specified an *identifier* instead. Possibly, you tried to create a pointer variable with improper declarations like the following:

```
VAR
    x : integer;
    y : ^x;
```

See the "Standard Pointer Type" section in Chapter 3 for details on setting up pointer types.

36 ERROR Multiple declaration of *variable* in parameter list.

You declared *variable* more than once in the parameter list of a procedure or function.

37 ERROR PROCEDURE/FUNCTION name required *(token)*.

You used the keyword **procedure** or **function** without specifying a valid identifier immediately after it. See the "Identifiers" section in Chapter 2 for a definition of a valid identifier. *Token* is either the name of the invalid identifier or the null set (if you did not supply any name at all).

38 ERROR Improper PROCEDURE/FUNCTION declaration *(token)*.

You were probably confusing procedure with function. A function has a data type, and a procedure does not; therefore, the following declaration triggers this error:

```
procedure one (r2 : single) : real;
```

39 ERROR

Improper parameter declaration *(token)*.

You did not specify the parameter list of your procedure or function in the correct manner. A common cause of this error is using semicolons incorrectly in the parameter list. (See Chapter 5 for details on parameter lists.) DOMAIN Pascal encountered *token* when it was expecting something else (probably a semicolon).

40 ERROR

Colon expected in FUNCTION declaration *(token)*.

You forgot to put a colon before the type specification of the function; for example, compare the right and wrong ways to declare a function:

```
FUNCTION pyth_theorem(a : integer16)   real;  {Wrong!}
FUNCTION pyth_theorem(a : integer16) : real;  {Right }
```

DOMAIN Pascal found *token* instead of a colon.

42 ERROR

FUNCTION type specification required.

You forgot to specify a data type for the function itself; for example, compare the right and wrong ways to declare a function:

```
{wrong} FUNCTION pyth_theorem(a : integer16);
{right} FUNCTION pyth_theorem(a : integer16) : real;
```

43 ERROR

CASE type is not scalar.

You specified an expression in a **case** statement that did not have a scalar data type. The scalar data types are integer, Boolean, char, enumerated, and subrange. For example, the following **case** statement triggers this error:

```
VAR
    r : real;

        . . .
BEGIN
   CASE r OF {error: r is real, but should have been
                     integer or integer subrange.    }
     1 : writeln('One');
     2 : writeln('Two');
   end;
```

44 ERROR *Constant* is not of the correct type for the CASE on line *number*.

You specified a *constant* in a **case** statement that was not the same data type as the expression of the **case** statement. For example, the following case statement triggers this error:

```
VAR
   r : integer16;

      . . .

CASE r OF
  1.5 : writeln('One');      {error: 1.5 is a real;
                              it should be integer.}
  2   : writeln('Two');
end;
```

45 ERROR *(Constant)* is outside the subrange of the CASE on line *number*.

In a CASE statement, you specified a *constant* that was not within the declared range of the **case**. For example, the following **case** statement triggers this error because the constant 5 is outside the declared subrange 0 to 3.

```
VAR
   x : 0..3;
BEGIN
   CASE x OF
      5 : . . . {error}
```

46 ERROR *Constant* has already occurred as a CASE constant on line *number*.

You specified the same *constant* more than once in the same **case** statement. For example, the following **case** statement triggers this error because constant 6 appears twice:

```
CASE r OF
  4     : writeln('square root is rational');
  5,6,7 : writeln('square root is irrational');
  6     : writeln('even');  {error}
end;
```

47 ERROR *Token* is not a valid option specifier.

You specified *token* in an OPTIONS clause, but *token* is not a valid option. See the "Routine Options" section in Chapter 5 for details.

48 ERROR Include file name must be quoted *(token)*.

You used the **%include** directive but forgot to put the name of the include file in apostrophes. *Token* is the token that DOMAIN Pascal found when it was expecting to find an apostrophe.

49 ERROR Too many include files.

Note that this error can be triggered by include files nested within include files. To correct this error, you can break the program into separately compiled modules.

50	ERROR	*Token* is not a recognized option.

You specified *token* as a compiler directive, but *token* is not a valid compiler directive. Refer to the "Compiler Directives" listing in Chapter 4 for a description. Possibly, you put a superfluous percent sign (%) in your program. Another possibility is that you specified *token* on your compile command line as a compiler option. If you do this, the operating system returns this error message. See Chapter 6 for a complete list of compiler options. Possibly, you caused this error by trying to compile two files at once, and the compiler interpreted the second file as an (invalid) option.

51 ERROR Include file *pathname* is not available.

You specified a *pathname* for an include file, but either it does not exist or network problems prevent the compiler from accessing it.

52 ERROR Semicolon expected following option specifier *(token)*.

You specified compiler directive **%debug** or **%eject** but forgot to specify a semicolon immediately after the directive. See the "Compiler Directives" listing in Chapter 4.

53 ERROR Multiple declaration of *identifier* in RECORD field list.

You specified the same field twice in a record declaration. For example, the following record declaration triggers this error because x is declared twice:

```
r = record
   x : Boolean;
   x : integer16;   {error}
end;
```

54 ERROR Array bound type is not scalar *(datatype)*.

You specified *datatype* as the index type of an array. However, *datatype* must be a scalar data type. The scalar data types are integer, char, enumerated, subrange, and Boolean.

55 ERROR Improper LABEL statement syntax *(token)*.

Labels must be unsigned integers or identifiers, but DOMAIN Pascal found *token* instead.

56 ERROR Multiple definition of *element*, previous definition was on line *number*.

You declared the same *element* (i.e., variable, data type, constant, label, procedure, or function) twice.

57 ERROR Improper usage of *identifier*, only a LABEL name is valid here.

You used *identifier* as a label, but you had already declared it as a variable, type, or constant. Possibly, you accidentally put a colon (:) immediately following the *identifier* in the action part of your program. The colon could cause the compiler to interpret the *identifier* as an illegal label. Perhaps you meant to specify a statement like the following:

```
x := 8;
```

but you forgot the equal sign and ended up specifying the following instead:

```
x : 8;
```

58 ERROR *Constant* is declared as a CONST name, and cannot be assigned a value.

You mistakenly tried to assign a value to a *constant*. Perhaps you should have declared *constant* as a variable rather than as a constant.

59 ERROR Improper use of *identifier*, only a VAR name is valid here.

You tried to assign a value to an *identifier* that is not a valid variable. Possibly, you tried to assign a value to a type rather than a variable. For example, code like the following causes this error:

```
TYPE
    int = integer32;
VAR
      q : int;
BEGIN
    int := 8;  {wrong}
      q := 8;  {right}
```

60 ERROR Improper use of *identifier*, only a VAR or CONST name is valid here.

You tried to assign the value of a data type or label to a variable. *Identifier* must be a variable or a constant.

61 ERROR *Token* is not an ARRAY.

You specified an expression of the format

```
TOKEN[. . .]
```

This format is reserved for specifying a particular element of an array; however, *token* is not an array variable. Possibly, you were trying to call a procedure or function and used brackets rather than parentheses.

62 ERROR *Variable* is not a pointer variable.

You tried to dereference a *variable* that was not declared as a pointer variable. (See the "Pointer Operations" listing of Chapter 4.)

63 ERROR *Token* is not a RECORD.

DOMAIN Pascal was expecting a **record** variable, but it found *token* instead. (See the "Record Operations" listing of Chapter 4.)

64 ERROR *Token* is not a field of *record*.

DOMAIN Pascal was expecting a field of the *record* variable, but it found *token* instead.

65 ERROR Too many subscripts to *array_variable*.

You declared *array_variable* as an n-dimensional array, but you have specified more than n subscripts for the array at this line.

66 ERROR *Kind_of_declaration* declaration must precede internal PROCEDURE and FUNCTIONS declarations.

You stuck a routine in the middle of a declaration part. A nested routine must come at the end of a declaration part (not in the beginning or the middle).

67 ERROR Improper use of *identifier*, only a FUNCTION name is valid here.

You probably had no intention of calling a function and are puzzled as to why you got this message. If you used a statement of the form

```
    IDENTIFIER(TOKEN)
```

then DOMAIN Pascal assumed that you were trying to call a function. Possibly, you were trying to access an array, but you used parentheses instead of brackets.

68 ERROR The types of *operand1* and *operand2* are not compatible with the *operator* operator.

You made a mistake such as specifying (21.0 DIV 3.0). (It's a mistake because **div** only accepts integer operands.) See Chapter 4 for a complete list of operators and their valid operands.

69 ERROR The type of *operand* is not compatible with the *operator* operator.

You made a mistake such as specifying an expression like:

```
    (NOT 3.0)
```

It's a mistake because **not** only accepts Boolean operands. See the beginning of Chapter 4 for a summary of operators.

70 ERROR Incompatible operands [*operand1,operand2*] to the *operator* operator.

See Table 4-1 for a summary of operators.

71 ERROR Subscript *expr* to array *name_of_array* is not of the correct type.

See the "Array Operations" listing in Chapter 4.

72 WARNING No path to statement *(name_of_statement)*, or no branch generated for it.

The program never reaches this statement; therefore, the compiler does not generate any code for it. Sometimes a **goto** statement triggers this warning.

73 ERROR *Statement* expression is not Boolean.

You were using a non–Boolean expression in a manner reserved for Boolean expressions. For example, the following program fragment triggers this error because variable `int` is an integer, not a Boolean:

```
VAR
    int : integer;
    b : Boolean;

        . . .

if int   then ...{error, int is not a Boolean expr. }
if int=9 then ...{no error, int=9 is a Boolean expr.}
if b     then ...{no error, b is a Boolean expr.    }
```

74 ERROR FOR statement index variable is not scalar.

The scalar data types are integer, Boolean, char, enumerated, and subrange. If you specify an index variable with a data type other than one of these five types, DOMAIN Pascal issues this error. See the **for** listing in Chapter 4.

75 ERROR FOR statement initial value expression is not compatible with the index variable.

You specified a start_expression of a different data type than the index variable. See the **for** listing in Chapter 4.

77 ERROR FOR statement limit value expression is not compatible with the index variable.

You specified a stop expression of a different data type than the index variable. See the **for** listing in Chapter 4.

79 ERROR Assignment statement expression is not compatible with the assignment variable.

You tried to assign the value of an expression to a variable, but the data type of the value and the variable were not compatible. In general, the data type of the expression must match the data type of the variable; however, there are a few exceptions. For example, you can assign an integer expression to a real variable (though you cannot do the reverse). In most cases, this error is just a simple programming mistake, but if you do intend to assign a value to a variable of a different data type, refer to the "Type Transfer Functions" listing of Chapter 4.

81	ERROR	Too many arguments to *routine*. You attempted to call *routine*, but you tried to pass more arguments to *routine* than it was expecting. The number of arguments cannot exceed the number of parameters declared in the parameter list of the *routine*. See Chapter 5 for details on parameter passing.
82	ERROR	Too few arguments to *routine*. You attempted to call *routine*, but you tried to pass fewer arguments to *routine* than it was expecting. If you want to pass n arguments to a routine declaring more than n parameters, you must use the **variable** routine attribute (which is described in the "Variable" section in Chapter 5).
83	ERROR	Argument *n* to *routine* is not compatible with the declared argument type. You tried to call *routine*, but the Nth argument in the call does not have the same data type as the Nth parameter. You can suppress this error by using the **univ** routine attribute. See Chapter 5 for details on parameter passing.
84	ERROR	Argument *n* to *routine* is not within the declared argument subrange. You tried to pass a subrange expression as the Nth argument to call *routine*, but the value of the expression was not within the declared range of the subrange.
85	ERROR	Improper use of *element*, only a PROCEDURE name is valid here. DOMAIN Pascal assumed you were trying to call a procedure, but *element* is not a procedure. Any statement having the following format is assumed to be a procedure call: `IDENTIFIER( anything );`
86	ERROR	Unrecognized statement *(token)*. The compiler could not classify a statement into one of the basic categories of DOMAIN Pascal statements (such as assignment, procedure call, function call, **goto**, **repeat**, etc.). Possibly, you misspelled a keyword, or perhaps you forgot to close the previous statement with a semicolon.
87	ERROR	GOTO label expected *(token)*. You forgot to specify a label immediately after the keyword **goto**. DOMAIN Pascal expected a declared variable, but found *token* instead. See the **goto** listing of Chapter 4.
89	ERROR	The value of *number* is outside the range of valid set elements. You tried to assign a *number* greater than 256 to a set. See the "Set Operations" listing in Chapter 4 for details on assigning values to sets, and see the "Sets" section in Chapter 3 for information on declaring set types and variables.

91 ERROR Function type must only be specified when the function is declared FORWARD.

You specified a function as **forward**, but you mistakenly specified the data type of the function twice. You must only specify the data type of the function once. Specify the data type when you specify **forward**. See the "Forward" section in Chapter 5 for an explanation of **forward**.

92 ERROR *(Option)* specifier is not valid when defining a procedure/function previously declared to be FORWARD.

If *option* is **forward**, then you probably declared **forward** twice for the same routine. Possibly, you declared a routine as **forward**, but you also used the routine with both **define** and **extern**. If *option* is **extern**, then you probably declared **extern** twice for the same routine.

93 ERROR Improper use of the DEFINE statement.

See Chapter 7 for a complete description of the **define** statement.

94 ERROR Improper DEFINE statement structure.

See Chapter 7 for a complete description of the **define** statement.

95 ERROR Multiple declaration of *element* in DEFINE statement.

You used **define** to define the same *element* twice. See Chapter 7 for a complete description of the **define** statement.

96 ERROR Constant value cannot be evaluated at compile time.

You specified an expression in a **const** declaration that the compiler could not reduce to a constant. For example, the following declarations trigger this error because x is not a constant:

```
VAR
     x : integer;

CONST
    ax : addr(x);
```

97 ERROR Label *label* is never defined.

You declared a *label* in the **label** declaration part of a routine, but you never specified this label inside the code portion of the routine. Possibly, you declared the label in the **label** declaration part of a routine, but specified this label inside the code portion of another routine. See Chapter 2 for a description of labels, and see the **goto** listing in Chapter 4 for a description of the **goto** statement.

98 ERROR Improper PROCEDURE/FUNCTION structure *(token)*.

Refer to Chapter 2 for the rules on routine structure. Possibly, you put a period instead of a semicolon at the end of a routine.

99	ERROR	BEGIN expected in routine *name_of_routine*; found "*token*".
		The code portion of a routine must start with the keyword **begin**. DOMAIN Pascal discovered *token* instead of **begin**. Refer to Chapter 2 for the rules on routine structure. Note that every routine (including the main program) must at least include the keywords **begin** and **end**.
100	ERROR	END expected; found "*token*".
		You forgot to mark the finish of a routine with an **end** statement. Ignore the line number the error is reported at; the compiler usually does not discover this error until the end of the program. See Chapter 2 for the rules on program structure.
101	ERROR	Statement separator expected *(token)*.
		DOMAIN Pascal discovered two statements with nothing to separate them. You probably made one of the following three mistakes. First, you forgot a semicolon. Second, you forgot a closing **end** in a compound statement. Third, you forgot an **else** in an **if/then/else** statement.
102	ERROR	Improper argument list *(token)*.
		You forgot to specify a ")" to terminate a type transfer function. See the "Type Transfer Functions" listing in Chapter 4.
103	ERROR	THEN expected in IF statement *(token)*.
		You forgot the **then** part of an **if/then/else** statement. For details on **then**, see the **if** listing in Chapter 4.
105	ERROR	OF expected in CASE statement (*(token)*).
		A **case** statement must begin with the format `CASE expr OF` but you forgot the keyword **of**. See the **case** listing in Chapter 4.
106	ERROR	CASE label expected *(token)*.
		In a **case** statement, you specified a statement without specifying a constant. Possibly, you forgot to conclude the **case** statement with **end**. See the **case** listing in Chapter 4.
107	ERROR	END/OTHERWISE expected in CASE statement *(token)*.
		You probably forgot to conclude a simple statement with a semicolon or a compound statement with an **end**.
109	ERROR	DO expected in WHILE statement *(token)*.
		You forgot to specify the keyword **do** following the condition in a **while** statement.

110 ERROR UNTIL expected in REPEAT statement *(token)*.

DOMAIN Pascal found *token* instead of **until** in a **repeat** statement. Refer to the **repeat** listing in Chapter 4.

111 ERROR := expected in FOR statement *(token)*.

DOMAIN Pascal found *token* instead of := in a **for** statement. Refer to the **for** listing in Chapter 4.

112 ERROR TO or DOWNTO expected in FOR statement *(token)*.

DOMAIN Pascal found *token* instead of **to** or **downto**. Refer to the **for** listing in Chapter 4.

113 ERROR DO expected in FOR statement *(token)*.

DOMAIN Pascal found *token* instead of **do**. Refer to the **for** listing in Chapter 4.

114 ERROR Improper WITH statement *(token)*.

Refer to the **with** listing in Chapter 4.

115 ERROR DO expected in WITH statement *(token)*.

Refer to the **with** listing in Chapter 4.

116 ERROR Improper expression *(expression)*.

A variety of situations could have caused this error. Probably, DOMAIN Pascal was expecting a keyword, and you either did not enter a keyword, or you did not enter a keyword that was appropriate to the situation. *Expression* is the inappropriate expression. For example, you can trigger this error by using the keyword **if** without using the keyword **then**. Another possibility is that you forgot a semicolon on the line preceding the line that the compiler reported the error. Another possibility is that you began an identifier with a digit or dollar sign ($) rather than a character.

117 ERROR Identifier expected *(token)*.

DOMAIN Pascal was expecting an identifier and found *token* instead. Chapter 2 defines identifiers.

120 ERROR OF expected in FILE declaration *(token)*.

You used the keyword **file** without following it with the keyword OF. See Chapter 3 for details on declaring file types.

121 ERROR Expression/constant cannot be passed as argument *n* to *routine*.

You specified an expression or constant as the *n*th argument to *routine*. However, the *n*th parameter of *routine* is declared as **var** or **in out**, and you can only pass variables as arguments to such a parameter.

122 ERROR Improper use of *identifier*.

You probably tried to call a predeclared procedure as a function or a predeclared function as a procedure. See Chapter 5 for a description of the difference between calling procedures and calling functions.

123 ERROR Attempted assignment to *(variable)*, a FOR–index variable, or formal parameter marked as IN.

You either tried to assign a value to *variable* inside a routine that declared it as **in**, or you tried to modify a FOR loop's index variable inside the loop. If you did the former, you can correct this error by changing the **in** parameter to **in out** or **var**. If your error was attempting to modify a **for** loop's index variable, you can eliminate the code inside the loop that modifies the variable.

124 ERROR *Routine* requires TEXT file parameter.

You specified a **file of** variable as an argument to *routine*, but *routine* requires a **text** variable instead.

125 ERROR *Procedure* requires FILE parameter.

You specified an illegal file parameter for **open** or **close**. Only identifiers are legal file parameters. Former FORTRAN programmers might have triggered this error by using an integer as a file parameter. Possibly, you forgot to specify any file parameter at all.

126 ERROR *Procedure* cannot be performed on INPUT file.

The standard input file (**input**) cannot be an argument to **rewrite**, **put**, or **page**. See the "Default Streams" section in Chapter 8 for details on **input**.

127 ERROR *Procedure* cannot be performed on OUTPUT file.

The standard output file (**output**) cannot be an argument to **reset**, **get**, **eof**, or **eoln**. See the "Default Streams" section in Chapter 8 for details on **output**.

128 ERROR Argument to *identifier* is not a pointer reference.

A predeclared procedure (typically **new** or **dispose**) requires an argument of a pointer type.

129 ERROR Operand *operand1* is not compatible with *(routine)*.

You tried to read a value into an expression; for example, consider the following statements:

```
READ(x + 1);              {wrong}
READLN(x + 1);            {wrong}
READLN(x); x := x + 1;    {right}
```

130 ERROR

Fraction width specified for operand *(element)* that is not of a REAL type.

In a **write** or **writeln** statement, you specified a two-part field width for a nonreal expression. If the expression is real, you can specify an optional one- or two-part field width, but if the expression is not real, then you can only specify an optional one-part field width. See the **write** listing in Chapter 4 for a complete description of field widths.

131 ERROR

Field width specifier is not permitted.

You can only specify a field width for a **write** or **writeln** statement. If you specify a field width for any other statement, DOMAIN Pascal issues this error. When you call a procedure or function, DOMAIN Pascal interprets any colon (:) inside the call as a field width.

132 ERROR

Field width specifier *(token)* is not INTEGER.

DOMAIN Pascal was expecting to find an integer field width, but found *token* instead. Remember, you format real numbers with a two-part field (not a decimal). Note that when you call a procedure or function, DOMAIN Pascal interprets any colon (:) inside the call as a field width. So, possibly you triggered this error with an inadvertent colon.

133 ERROR

Type of operand *(identifier)* is not compatible with the *routine* operation.

You tried to read or write an aggregate variable (such as an array or record) to or from a text (UASC) file. You can correct this mistake by specifying a rec file instead of an UASC file. If you must use a text file, you can correct the error by specifying a field (if a record) or an element (if an array) rather than the full aggregate.

135 ERROR

Improper file mode in OPEN.

The file mode is the third argument to the predeclared **open** procedure. The file mode must be a character string, a string constant, or a variable whose data type is an array of char. See the **open** listing in Chapter 4 for details on file modes.

136 ERROR

Improper status argument in *procedure*.

You specified a status argument to *procedure* that had a data type other than **integer32**. A common mistake is to misuse a `status_$t` variable. For example, compare the right and wrong ways to use such a variable in an **open** procedure:

```
%INCLUDE '/sys/ins/base.ins.pas';
   VAR
       st     : status_$t;
       f1, f2 : text;
      . . .
   OPEN(f1, 'anger1', 'NEW', st);      {wrong}
   OPEN(f2, 'anger2', 'NEW', st.all);  {right}
```

Refer to the **open** and **find** listings in Chapter 4 for details on syntax.

137 ERROR *Procedure* cannot be used on a text file.

The first argument to **find** or **replace** must be a variable of type **file**. You have mistakenly specified a variable of type **text**. Refer to the **find** or **replace** listings in Chapter 4 for details.

138 ERROR Record number is not integer in FIND.

The second argument to the **find** procedure is the record number, and this argument must be an integer expression. Refer to the **find** listing in Chapter 4 for details on syntax.

139 ERROR Improper parameter list in PROGRAM statement ^1.

You made a mistake in your program heading. If you specified a file list in the program heading, then make sure that you specified the files as identifiers (and not as strings). For example, compare the following two file lists:

```
PROGRAM test(input, output);     {right}
PROGRAM test('input', 'output'); {wrong}
```

140 ERROR Compiler failure, unknown tree node.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

141 ERROR Compiler failure, unknown top node.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

142 ERROR Compiler failure, no temp space.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

143 ERROR Compiler failure, lost value of node.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

144 ERROR Compiler failure, registers locked.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

145 ERROR Compiler failure, no emit inst.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

146 ERROR Compiler failure, procedure too large.

The routine you are trying to compile may have exceeded compiler implementation limits. Try to break the routine into multiple routines and modules and then recompile. If the problem still persists, please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

147 ERROR Compiler failure, inst disp too large.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

148 ERROR Compiler failure, obj module too large.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

149 ERROR Compiler failure, no free space.

The compiler ran out of dynamic memory while compiling your program. Try to break the program into multiple routines and modules and then recompile. If the problem still persists, please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

150 ERROR Compiler failure, short branch optimization.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

151 ERROR *Routine* was declared FORWARD on line *number* and never defined.

You specified *routine* as **forward**, but you did not define it in your program. See the "Forward" section in Chapter 5 for details on **forward**.

152 ERROR Section name *(identifier)* conflicts with procedure or data section name.

You specified *identifier* as a section name for a **var** declaration part; however, *identifier* is a reserved section name. Please pick another name instead, or remove the section name completely. See the "Putting Variables Into Sections" section in Chapter 3 for details on naming sections.

153	ERROR	Improper section name specification *token*. You were declaring a section name for a group of variables, but you specified *token* rather than an identifier as the name of the section. See the "Putting Variables Into Sections" section in Chapter 3 for details on naming sections.
154	ERROR	Conflicting storage allocation specifications. You declared a variable as **static** and as belonging to a nondefault section name. It cannot be declared as both at the same time.
155	WARNING	Constant subscript *(value_of_constant)* to array *name_of_array* is out of range. The *value_of_constant* was not within the declared range of *name_of_array*. You must either use a different constant or expand the declared range of the array. See the "Arrays" section in Chapter 3 for a description of array declaration.
157	ERROR	*Identifier* was declared in a DEFINE statement but never defined. You used *identifier* in a **define** statement, but forgot to use it as the name of a procedure, function, or variable. See Chapter 7 for an explanation of the **define** statement.
158	ERROR	Improper OPTIONS specification *(token)*. You made a mistake while declaring OPTIONs for a routine. Probably, you specified *token* instead of a valid routine attribute. Possibly, you forgot to mark the end of an OPTIONs clause with a semicolon. The valid routine attributes are listed in Chapter 5.
159	ERROR	Duplicate OPTIONS specification *(routine_attribute)*. You specified the same routine attribute twice in an OPTIONs clause. You can only specify it once. The "Routine Options" section in Chapter 5 describes the OPTIONs clause.
160	ERROR	Conflicting OPTIONS specification *(routine_attribute)*. You specified several routine attributes in an OPTIONs clause; however, *routine_attribute* cannot appear in the same OPTIONs clause as one of the previous routine attributes. For example, you cannot specify both **forward** and **extern** in the same OPTIONs clause. You can also trigger this error by mistakenly using the same routine attribute twice. The "Routine Options" section in Chapter 5 describes the OPTIONs clause.
161	ERROR	Unrecognized OPTIONS specification *(token)*. You specified *token* inside an OPTIONs clause, but *token* is not a valid routine attribute (described in Chapter 5).

162	WARNING	Conditional compilation user warning. You triggered a warning–level problem through misuse of the conditional compiler directives. More specific messages will follow this one. See the "Compiler Directives" listing of Chapter 4 for details on the conditional compilation directives.
163	ERROR	Conditional compilation user error. You triggered an error–level problem through misuse of the conditional compiler directives. More specific messages will follow this one. See the "Compiler Directives" listing of Chapter 4 for details on the conditional compilation directives.
164	ERROR	Conditional compilation syntax error; look at prior "(PreProc)" message. "(PreProc)" is an abbreviation for the DOMAIN Pascal preprocessor. This error is telling you that the preprocessor found an error and passed it along to the compiler. The preprocessor found an error in a conditional compilation directive. (The conditional compilation directives are **%var**, **%if**, **%then**, **%else**, **%config**, **%elseif**, **%elseifdef**, **%enable**, **%endif**, and **%ifdef**.) Possibly, you used a conditional compilation variable without having first declared it (with a **%var** directive). Another possibility is that you used an operator other than **and**, **or**, and **not** in a predicate. See the "Compiler Directives" listing in Chapter 4 for details.
165	ERROR	Conditional compilation not balanced. Probably, you forgot to end an **%if** directive with the **%end** directive. (Making this mistake may trigger several other errors including Error 6: "Period expected at end of program.") See the "Compiler Directives" listing in Chapter 4 for details.
166	ERROR	Compiler failure, data frame overflow. The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)
168	ERROR	Compiler failure, register consistency. The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)
169	ERROR	Compiler failure, no temp created. The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)
170	ERROR	Compiler failure, improper forward label at *token*. The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

171	ERROR	Compiler failure, pseudo pc consistency.
		The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)
173	ERROR	Cannot take the address of internal routine *identifier*.
		You cannot pass *identifier* as an argument to the **addr** function, because it is an internal routine. An internal routine is any routine declared in the main program, any routine nested inside another routine, or any routine specified with the routine option **internal**.
174	ERROR	Ptr is *keyword1* type but operand is a *keyword2*.
		You tried to assign a value that has a data type other than what the pointer is expecting. Remember that in DOMAIN Pascal, unless you use **univ_ptr**, a pointer can only point to a value of the specified data type. See Chapter 3 for a discussion of pointer types.
175	ERROR	Incompatible function return types.
		A pointer to a function is expecting a value of a particular data type to be returned to it, but you mistakenly tried to return a value of a different data type to it. See Chapter 3 for a discussion of pointer types.
176	ERROR	Incompatible VARIABLE arguments options.
		Possibly, there is a mismatch between a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.
177	ERROR	Incompatible number of parameters.
		Possibly, there is a mismatch between a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.
178	ERROR	Incompatible ECB options.
		Possibly, there is a mismatch between a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.
179	ERROR	Incompatible parameter passing conventions for parameter *identifier*.
		Possibly, there is a mismatch between the parameter–passing conventions declared for a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.
180	ERROR	Incompatible types specified for parameter *identifier*.
		Possibly, there is a mismatch between the parameter–passing conventions declared for a pointer to a procedure or function, and the pointer value you are actually trying to assign to it.

181	ERROR	INTERNAL option is illegal for PROCEDURE^ or FUNCTION^ types. You mistakenly tried to use the routine option **internal** in a procedure or function pointer.
183	ERROR	"[" expected; "*token*" found. The compiler was expecting to encounter a left bracket "[", but found *token* instead.
184	ERROR	"]" expected; "*token*" found. The compiler was expecting to encounter a right bracket "]", but found *token* instead.
185	ERROR	Illegal type of constant "*token*" for variable "*identifier*". You were trying to initialize variable *identifier,* but you mistakenly specified a value *token* that did not have the same type as *identifier*. The compiler will not perform automatic type transfers for variable initializations in the **var** declaration part.
188	ERROR	Dynamic variable *identifier* cannot be initialized. By default, all variables declared in routines other than the main program will be allocated dynamically. You cannot initialize dynamic variables in the **var** declaration part. If you want to get around this problem, you can use the variable allocation clause **static** to force the compiler to store a routine variable nondynamically (i.e., statically). If you use **static,** the compiler lets you initialize the variable.
190	ERROR	Cannot initialize null array *identifier.* You specified a null array (i.e., an array that takes up no space in main memory) which by itself would only cause a warning; however, you mistakenly tried to initialize the null array.
191	WARNING	String initializer too long for *name_of_array*; truncated to fit. You tried to initialize *name_of_array* with a string that had too many characters. The compiler is warning you that you lost one or more characters of the string in the initialization. To avoid this, you should probably use an asterisk in the index expression of the array. The asterisk tells the compiler to figure out how many characters the string requires and declares the array accordingly. See the "Defaulting the Size of an Array" section in Chapter 3 for details.
192	ERROR	Variable *name* is not EXTERN; cannot DEFINE it. You declared *name* in a **var** declaration part, and you tried to define it in a **define** statement. You can only specify *name* in a **define** statement if you also declare it as an **extern** variable. (See Chapter 7 for details.)

193	WARNING	Space filling is NOT done for arrays greater than *number* bytes.

You are assigning all the elements of a string to a larger array of char. The compiler is warning you that the remaining elements in the larger array may have garbage values in them. That is, it won't blank pad the remaining elements in the array.

194 WARNING Unbalanced comment; another comment start found before end.

You specified two comment start delimiters without specifying a comment end delimiter in between them. You can suppress this warning with the **-ncomchk** compiler option (described in Chapter 6.) Refer to the "Comments" section in Chapter 2 for details on comments.

195 ERROR Lower bound must be an integer value for upper bound of "*".

You used an asterisk (*) to force the compiler to determine the number of elements in the array, but you mistakenly specified a noninteger value as the lower bound of the array. For example, consider the right and the wrong way to use the asterisk:

```
VAR
     x : array[ 1..* ] of char := 'HELLO';   {right}
     x : array['a'..*] of char := 'HELLO';   {wrong}
```

196 WARNING Size of *array* is zero.

You specified an array whose index makes no sense. For example, you specified an enumerated value for the lower bound and an asterisk for the upper bound. (See Chapter 3 for details on array declaration.)

197 ERROR Illegal repeat count usage; valid for array elements only.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3.

198 ERROR Illegal type for repeat count *(token)*; must be integer.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

199 ERROR Illegal repeat count value *(token)*; must be greater than zero.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

200 ERROR Repeat count too large by *number* for array.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

201 ERROR OF expected for repeat count.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

202 ERROR Illegal use of "*" repeat count for variable array *identifier*.

See the "Using Repeat Counts to Initialize Arrays" section in Chapter 3 for details on using repeat counts.

203 ERROR ":=" expected in record initialization.

You were trying to initialize a record field with a value, but you forgot the assignment phrase (:=). Perhaps you mistakenly used (=) instead of (:=).

204 ERROR Too many initializers for record init; field list exhausted at constant *value_of_constant*.

If a record has n fields, you tried to initialize more than n fields. You can only initialize n or less than n fields. See the "Initializing Data in a Record" section in Chapter 3 for details.

205 ERROR Size of *type_transfer_function* is not the same as the size of *datatype*.

You misused a type transfer function. The size (in bytes) of the *datatype* and the *type_transfer_function* must be equal. Chapter 3 details the sizes of all data types.

206 ERROR := expected in assignment statement *(token)*.

Probably, you made a mistake on the left side of an assignment statement that involved a type transfer function.

207 ERROR Constant cannot be passed as argument *n* to *routine*.

You tried to pass a constant as the *n*th argument to *routine;* however, the *n*th parameter in *routine* was declared as **var, out,** or **in out**. There are three ways to get around this problem. First, you can change **var, out,** or **in out** to **in**. Second, change **var, out,** or **in out** to a value parameter. Third, change the constant to a variable. See Chapter 5 for a complete explanation of parameters.

208 ERROR Expression (operator = *token*) cannot be passed as argument *n* to *routine*.

You mistakenly tried to pass an expression as the *n*th argument to *routine*. The problem is that the *n*th parameter of *routine* is a **var, out,** or **in out** parameter. You should probably change the parameter to become a value parameter. See Chapter 5 for a complete explanation of parameters.

209 WARNING Large (*number_of_bytes* bytes) copy of argument *name_of_arg* will be done when *routine* is invoked.

You are trying to pass a large data structure (probably an array) as a value parameter. This is going to take up a lot of CPU time at runtime. You should change the value parameter to a variable, **in** or **out** parameter. Chapter 5 describes the various kinds of parameters.

210	WARNING	Routine *name_of_routine* needs *number* bytes of stack, which approaches the maximum stack size of *max_size* bytes. You are trying to pass a large data structure (probably an array) as a value parameter. Consequently, your program will probably execute quite slowly. You should change the value parameter to a **var**, **in**, **out**, or **in out** parameter. The "Parameter Types" section in Chapter 5 describes all the parameter types.

211	WARNING	Routine *name* needs *number* bytes of stack, which exceeds the maximum stack size of *max_size* bytes.
		
		You probably have a large data structure (usually an array) in your code, and you may be trying to pass the structure as a value parameter. For example, an array like this
		
		`VAR`
		`    big_array : array[1..100000] of integer32;`
		
		might exceed the maximum stack size.
		
		If you try to run the program, you will probably get an "access violation" error. If the structure is a value parameter, you should change it to a **var**, **in**, **out**, or **in out** parameter. The "Parameter Types" section in Chapter 5 describes all the parameter types.
		
		This warning can occur when you compile a program on one type of workstation, but not occur when you compile on another type. For example, your program might work fine on a DN460 but when you compile it on a DN330 this warning might occur. This is because of the difference in virtual address space available on different nodes.

212	ERROR	Function *name* returns more than 32K bytes. The data type of the function consumes more than 32K bytes of memory. Probably, the data type of the function is a large array. Instead of passing the information back through the function, you should pass it back through a parameter.
213	ERROR	Illegal FOR statement index variable; *identifier* is a record or an array reference. DOMAIN Pascal does not permit a component of a record or an element of an array to be the index–variable in a **for** statement.
214	ERROR	Size of argument *n* to *routine* is not equal to the expected size of *number* bytes. You tried to pass a string as the *n*th argument to *routine*, but the *n*th parameter of *routine* was expecting a larger or smaller string. You must either change the size of the argument to match the size of the parameter, or you must declare the parameter as **univ**. See the "Univ" section in Chapter 5 for details.

228	ERROR	Too many initializers for array init; "]" expected, "*token*" found. You specified more data for the array than the array can hold. (See Chapter 3 for details on declaring arrays.)
234	ERROR	Compiler failure, too many nodes. This program is so large that the compiler cannot optimize it. You can try recompiling with **-opt 0** (see Chapter 6), but we recommend that you reduce the size of the program by breaking it up into modules. Chapter 7 explains modules.
235	WARNING	Potential illegal use of FOR index variable *(identifier)* outside of FOR stmt. DOMAIN Pascal forbids the use of the value of the index-variable after normal termination of a **for** loop. The compiler generates this message if a **for** loop has no premature exits (**exit** or **goto**) and the value of the index-variable is used outside the loop.
236	ERROR	Floating-point constant "*number*" conversion problem. *Number* was so large that the compiler encountered an overflow error when it tried to convert it from a double to a single, from a double to an integer, or from a single to an integer.
237	ERROR	Compiler failure, unexpected data init construct: *token.* The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)
238	ERROR	*Type_of_routine1 identifier* was previously declared as a *type_of_routine2.* You specified *identifier* as a forward procedure, but in the routine heading, you specified it as a function. Or, you specified *identifier* as a forward function, but in the routine heading, you specified it as a procedure. You must declare it as a procedure in both places or as a function in both places.
240	WARNING	Size of constant *(value)* is greater than the number of bits *(number)* in packed field *name*; constant has been truncated. *Value* is outside the declared subrange of *name*. You must specify a *value* that falls within the declared range, redeclare *name*, or omit the keyword packed from the record declaration. (See the "Records" section in Chapter 3 for details on space allocation in packed and unpacked records.)
241	ERROR	Dividing by zero in a compile-time constant expression. You, following the same mistaken path that Einstein once trod, have tried to divide by zero.

242	ERROR	Size of a PROCEDURE or FUNCTION is undeterminable.
		You mistakenly specified the name of a routine as an argument to the **sizeof** function. See the **sizeof** listing in Chapter 4 for a list of its legal arguments.
243	WARNING	Variable *name* was not initialized before this use.
		The compiler is warning you of the possibility of a garbage result when using the value of variable *name*. To solve this problem, you must assign a value to *name*. If *name* was declared as an **out** parameter, then you should probably change it to an **in out** parameter.
244	WARNING	UNIV parameter *name* should not be passed as a value-parameter.
		You specified a **univ** parameter as a value parameter. You should explicitly declare **univ** parameters as **in, out, in out**, or **var**. (See the "Univ" section in Chapter 5 for details on **univ**.) At runtime, the called routine copies the value parameter. Since the site of the parameter and the argument might differ, using a **univ** value parameter might cause runtime problems.
245	ERROR	Compiler Failure, Store Elimination Error
		The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)
246	WARNING	Expression passed to UNIV formal *name* was converted to *newtype*.
		See the "Univ" section in Chapter 5 for details on this warning message.
247	ERROR	Compiler failure, implementation restriction: Identifier-list contains too many names.
		This is an implementation restriction. You specified too many identifiers in an enumerated type. (See the "Enumerated Data" section in Chapter 3 for details on declaring enumerated types.)
248	ERROR	Compiler failure, limit exceeded; *limitation_message*.
		The *limitation_message* explains the problem.
249	ERROR	Too many nested pointer references for debug tables.
		This is an implementation restriction. The symbol table (used by the debugger) cannot process a record containing pointers to other records in a chain longer than 256 elements.
250	WARNING	*Name_of_statement* statement was constant-folded at compile time.
		Several conditions can cause the compiler to constant-fold a statement. One of the more common is because the DOMAIN Pascal compiler (SR9 or later) recognizes an attempt to compare a negative number to an unsigned subrange variable. When it recognizes such an attempt, it optimizes the code and issues a warning message. For example, consider the following fragment:

```
VAR
    x : 0..65535;

BEGIN
            . . .
    IF x = -1 THEN RETURN;
            . . .
END;
```

The compiler generates *no* code for the **if/then** statement because it knows that a negative value of x is not possible.

When the compiler notes such a contradiction, it issues warning messages. These messages come from the following three groups:

```
(IF|WHILE|CASE) statement was constant-folded at
    compile-time.

Comparison is false
Comparison is true

<= becomes =
> becomes <>
>= becomes =
< becomes <>
```

For example, if you compile the following program:

```
Program warning_test;
VAR
   x : 0..100;
BEGIN
   write('Enter an integer -- '); readln(x);
   if x <= 0 then writeln('hi');
END.
```

The compiler issues the following two warning messages:

```
<= becomes =
      and
IF statement was constant-folded at compile-time.
```

The first message tells you that the compiler is going to optimize the **if/then** statement. The second message tells you that the compiler is going to code the <= as an = because a < condition is not possible.

If you write the **if/then** statement in the program as

```
if x < 0 then writeln('hi');
```

the compiler prints a "Comparison is false" warning message because it is apparent to the compiler that there is no way that x < 0 can ever be true. In such a case, the compiler generates *no* code for the **then** part of the statement.

251	ERROR	Conflicting use of section name (*name_of_section*). You specified *name_of_section* as both a code section name and a data section name. It cannot be both. See the "Section" section in Chapter 5 for details.
252	ERROR	Compiler failure, invalid use of multiple sections and non-local goto to label *name_of_label*. The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)
253	ERROR	Compiler failure, bad address constant. The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)
254	ERROR	Compiler failure, invalid use of multiple sections and up-level referencing in routine ^1. The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)
255	WARNING	<= becomes =. See the description of WARNING message number 250.
256	WARNING	> becomes <>. See the description of WARNING message number 250.
257	WARNING	Comparison is false. See the description of WARNING message number 250.
258	WARNING	Comparison is true. See the description of WARNING message number 250.
259	WARNING	>= becomes =. See the description of WARNING message number 250.
260	WARNING	< becomes <>. See the description of WARNING message number 250.

262 WARNING

Value-parameter *name* was not specified in call to FUNCTION or PROCEDURE *identifier* declared OPTIONS(VARIABLE).

You forgot to specify a value for the argument corresponding to parameter *name*. You might get runtime access violations since the called procedure copies value parameters into temporary storage, and the procedure has a variable number of parameters. You can correct this problem in one of two ways. You can assign a value to the argument, or you can change the value parameter to a **var**, **in**, or **in out** parameter.

263 ERROR

Only records may have variant tags.

Variant tags give you the capability to create records with variable sizes. For example, consider the following:

```
TYPE
   emp_stat      = (exempt, nonexempt);
   workerpointer = ^worker;
   worker = record
      first_name : array[1..10] of char;
      last_name  : array[1..14] of char;
      next_emp   : workerpointer;
      CASE emp_stat OF
         exempt    : (salary : integer16);
         nonexempt : (wages  : single;
                      plant  : array[1..20] of char);
   end;

VAR
   current_worker : workerpointer;
```

The `emp_stat` field is a variant tag field because it uses different amounts of storage depending on its value. The function **sizeof** and the procedures **new** and **dispose** can use variant tags -- for example, `NEW(current_worker, exempt)` -- but only when such tags are part of a **record** variable. This error occurs if you try to use a variant tag that is not part of a **record**.

264 ERROR

Too many variant tags specified for record.

You used more variant tags in the routines **new**, **dispose**, and **sizeof** than are present in the record type variable. (See the "Variant Records" section of Chapter 3 for a description of variant tags.)

265 ERROR

Type of tag *name* incompatible with variant.

The value you supplied when specifying a variant tag field is not one of the choices listed in the field declaration of the record variable. (See the "Variant Records" section of Chapter 3 for a description of variant tags.)

266	ERROR	No variant with value of tag *name* exists.

The value you supplied for a variant tag in the routines **new**, **dispose**, or **sizeof** is not one of the choices listed in the field declaration of the record variable. For example, you would get this error if you used the record declaration listed at ERROR message number 263, and then included this line in your program:

```
NEW(current_worker, salaried)
```

The error would occur because `salaried` is not one of the choices for the variant tag emp_`stat`. (See the "Variant Records" section of Chapter 3 for a description of variant tags.)

267	WARNING	*Token1* should not be followed by *token2*; the *token1* will be ignored.

This usually appears when you have a misplaced semicolon. You might have put a semicolon (*token1*) before the reserved word **else** (*token2*).

268	WARNING	Missing operator or statement terminator; inserted *token* to continue parsing.

The compiler generates this message when it is attempting to recover from errors and so continue parsing. It acts as if the missing *token* (usually a semicolon) were present, generates this message, and then goes on. To eliminate this message, insert the necessary delimiter(s) in your program and recompile.

269	ERROR	Variables in libraries must be external.

The variables declared in a precompiled library file must be accessible to a program that uses the file. However, if your precompiled library contains **static** and/or **define** variables, the calling program cannot access those variables because they are not explicitly external. Such variables are not permitted. To eliminate this error, eliminate the **static** or **define** identifier from the library precompilations.

270	ERROR	Compiled library failure, illegal object type.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us a UCR. (You can use the Shell command **crucr** to create a UCR form.)

271	ERROR	File is not a library *pathname*.

Pathname, which is supposed to specify a precompiled source library file, either is not a precompiled library at all, or is a precompiled library file whose data has been corrupted. Verify *pathname*. If it is incorrect, make the appropriate fix to your code. If it is correct, precompile it again to try to get rid of the corrupted data.

272	ERROR	*Library* is incompatible because it was generated by a more recent version of compiler.

Library files are not guaranteed to be forward compatible. For example, if you use libraries that the newest compiler produced in a program you compile with an older compiler, the two may be incompatible. To correct this you can use the newer compiler to compile your source program, or use an older compiler to produce the precompiled libraries.

273	ERROR	Bodies of PROCEDUREs/FUNCTIONs may not be declared in LIBRARY MODULES.

The routines defined in a precompiled library file must be accessible to a program that uses the file. However, such routines are not accessible unless they are marked with the **extern** attribute (described in Chapter 7). A routine in your precompiled library file was not marked **extern**.

274	ERROR	FORWARD PROCEDURE/FUNCTION declarations are not allowed in LIBRARY MODULES.

Forward declarations of procedures or functions are not allowed in a precompiled library file because such routines are not accessible to a program that uses the file. Rewrite your code to eliminate the **forward** declaration and recompile.

276	ERROR	":" not permitted after OTHERWISE.

You put a colon (:) after the keyword **otherwise** in a DOMAIN Pascal **case** statement. **Otherwise** is a clause, not a label, so it does not take a colon.

277	ERROR	Labels not permitted at MODULE level.

Since a module (described in Chapter 7) consists of named routines only, a label can only be declared within the scope of one of those routines. That is, there is no "main program" in a module with which a label at module level can be associated. However, you declared a label at module level. To correct this error, declare the label within the scope of the **program** block, or inside one of the routines in the module.

278	WARNING	*Identifier* has already been used in another context in current scope.

Pascal forbids the redeclaration of a name that has already been used within a program block. For example, the following code fragment declares a constant `ten` at program level. Within procedure `bo`, `ten` is used to initialize variable `hold`. `Ten` then is illegally redeclared as a variable of type **real**.

```
PROGRAM illegal;
CONST ten := 10;
PROCEDURE bo;
   VAR hold : integer := ten;
       ten  : real;            { This is illegal! }
        .
        .
        .
```

279 WARNING

Value assigned to *identifier* is never used; assignment is eliminated by optimizer.

If *identifier*'s value has side effects, such as in a function call or in a reference to variables with the **device** attribute, the value still is computed, and the optimizer only eliminates the assignment to *identifier*. However, if there are no side effects, the optimizer also eliminates the value's computation.

In most cases, you can simply eliminate the value assignment to *identifier* to get rid of this warning. However, there are times when you need to call a function, but are not interested in the value it returns and so don't use that value. In that case, use the **discard** function to explicitly eliminate the value assignment. See Chapter 4 for a description of the **discard** function and Chapter 3 for information about the **device** attribute.

280 WARNING

Current semantics for subrange of CHAR is incompatible with SR9.

Earlier versions of the compiler (SR9 and before) incorrectly use 16 bits to store a subrange of **char**, but SR9.5 only uses eight bits to store the subrange. The difference in the number of bits the compiler versions use can introduce incompatibilities among compilation units and data files. Therefore, this warning message has been added for the SR9.5 release only to indicate that a potential source of trouble exists.

281 FATAL

Too many compilation errors -- compilation terminated.

The errors in your program have caused an access violation in the compiler and so compilation cannot continue. Correct the problems already indicated and recompile.

283 ERROR

EXTERN PROCEDURES/FUNCTIONS may not be DEFINED in PROGRAM *name*.

A main program (which contains the **program** heading) may reference externally declared routines, but it may not **define** any global entry points. Your program tried to define one or more such points.

284 ERROR

FILE parameters may not be passed by value *paramname*.

Pascal forbids passing a file variable as a value parameter. To eliminate this error, change *paramname*'s declaration to **var**. The "Parameter Types" section in Chapter 5 describes all the parameter types.

285 ERROR

GOTO transfers control to a structured statement outside of its scope *token*.

Your code includes an erroneous **goto** into a structured statement. Structured statements include **case**, **while**, **repeat**, **for**, and **with**.

286 ERROR

Maximum line length (argument 5 to OPEN) must be an INTEGER.

You gave a value that is not an **integer** for the buffer_size argument to the **open** statement. The value must be an **integer**. See the listing for **open** in Chapter 4 for more details.

287	ERROR	Maximum line length (argument 5 to OPEN) may only be specified for TEXT files. An **open** statement may only include the buffer_size argument if you are opening a **text** file. However, you included the argument for an **open** of some other file type.
288	ERROR	Base types of *token1* and *token2* are incompatible. This error occurs if you try to use the **pack** or **unpack** built–in procedures on two arrays that have different base data types. Those types must be the same. For example, if one is an array of **integer32**, the other must be an array of **integer32**.
289	WARNING	Must overflow bounds of array (*token1*) in order to match PACKED array. This error can occur if you are using the **pack** or **unpack** built–in procedures. For every element in the packed array there must be a corresponding element in the unpacked array. See the listings for **pack** and **unpack** in Chapter 4 for more details.
294	ERROR	PROCEDURE may not be called in this context (*token*). It is illegal to use a procedure name (*token*) in the argument list of a call to another routine. This is because all of a routine's arguments must have or resolve to values, and while a function returns a value, a procedure does not.
295	ERROR	Modulus must be >= zero (*token*). This error can only occur if you compile with the **–iso** switch. The error occurs if the compiler can detect that the result of a **mod** function will be negative. The modulus must be greater than or equal to zero.

Appendix A

Reserved Words and Predeclared Identifiers

This appendix lists the reserved words and predeclared identifiers in DOMAIN Pascal.

Reserved words, listed in Table A–1, are names of statements, data types, and operators. You can use reserved words only with their reserved meanings (and within strings and comments). You cannot use a reserved word as an identifier.

Table A–1. Reserved Words

and	**end**	**not**	**set**
array	**file**	**of**	**then**
begin	**for**	**or**	**to**
case	**function**	**packed**	**type**
const	**goto**	**procedure**	**until**
div	**if**	**program**	**var**
do	**in**	**record**	**while**
downto	**label**	**repeat**	**with**
else	**mod**		

Table A–2 lists the predeclared identifiers. These identifiers name types, functions, procedures, values, and files. You can redefine predeclared identifiers; however, doing so means that you can no longer use the identifier for its original meaning within the scope of the redefinition.

Table A–2. Predeclared Identifiers

abs	false	next	rshft
addr	find	nil	set_sr
arctan	firstof	odd	sin
arshft	forward	open	single
boolean	get	ord	sizeof
char	in_range	otherwise	sqr
chr	input	out	sqrt
close	integer	output	static
cos	integer16	page	string
define	integer32	pred	succ
disable	internal	put	text
discard	lastof	read	true
dispose	ln	readln	trunc
double	lshft	real	univ
enable	max	replace	univ_ptr
eof	maxint	reset	val_param
eoln	min	return	write
exit	module	rewrite	writeln
exp	new	round	xor
extern			

Appendix B

ASCII Table

DOMAIN Pascal uses the ASCII character set for representing character data. Table B–1 shows the decimal, octal, and hexadecimal values for all ASCII characters.

Table B–1. ASCII Codes

oct	*dec*	*hex*	*character*		*oct*	*dec*	*hex*	*character*
0	0	0	NUL	^@	40	32	20	space
1	1	1	SOH	^A	41	33	21	!
2	2	2	STX	^B	42	34	22	"
3	3	3	ETX	^C	43	35	23	#
4	4	4	EOT	^D	44	36	24	$
5	5	5	ENQ	^E	45	37	25	%
6	6	6	ACK	^F	46	38	26	&
7	7	7	BEL	^G	47	39	27	’
10	8	8	BS	^H	50	40	28	(
11	9	9	TAB	^I	51	41	29	)
12	10	A	LF	^J	52	42	2A	*
13	11	B	VT	^K	53	43	2B	+
14	12	C	FF	^L	54	44	2C	,
15	13	D	CR	^M	55	45	2D	–
16	14	E	SO	^N	56	46	2E	.
17	15	F	SI	^O	57	47	2F	/
20	16	10	DLE	^P	60	48	30	0
21	17	11	DC1	^Q	61	49	31	1
22	18	12	DC2	^R	62	50	32	2
23	19	13	DC3	^S	63	51	33	3
24	20	14	DC4	^T	64	52	34	4
25	21	15	NAK	^U	65	53	35	5
26	22	16	SYN	^V	66	54	36	6
27	23	17	ETB	^W	67	55	37	7
30	24	18	CAN	^X	70	56	38	8
31	25	19	EM	^Y	71	57	39	9
32	26	1A	SUB	^Z	72	58	3A	:
33	27	1B	ESC	^[	73	59	3B	;
34	28	1C	FS	^|	74	60	3C	<
35	29	1D	GS	^]	75	61	3D	=
36	30	1E	RS	^^	76	62	3E	>
37	31	1F	US	^_	77	63	3F	?

Table B-1. ASCII Codes (continued)

oct	dec	hex	character	oct	dec	hex	char
100	64	40	@	140	96	60	'
101	65	41	A	141	97	61	a
102	66	42	B	142	98	62	b
103	67	43	C	143	99	63	c
104	68	44	D	144	100	64	d
105	69	45	E	145	101	65	e
106	70	46	F	146	102	66	f
107	71	47	G	147	103	67	g
110	72	48	H	150	104	68	h
111	73	49	I	151	105	69	i
112	74	4A	J	152	106	6A	j
113	75	4B	K	153	107	6B	k
114	76	4C	L	154	108	6C	l
115	77	4D	M	155	109	6D	m
116	78	4E	N	156	110	6E	n
117	79	4F	O	157	111	6F	o
120	80	50	P	160	112	70	p
121	81	51	Q	161	113	71	q
122	82	52	R	162	114	72	r
123	83	53	S	163	115	73	s
124	84	54	T	164	116	74	t
125	85	55	U	165	117	75	u
126	86	56	V	166	118	76	v
127	87	57	W	167	119	77	w
130	88	58	X	170	120	78	x
131	89	59	Y	171	121	79	y
132	90	5A	Z	172	122	7A	z
133	91	5B	[	173	123	7B	{
134	92	5C	\	174	124	7C	|
135	93	5D	]	175	125	7D	}
136	94	5E	^	176	126	7E	~
137	95	5F	_	177	127	7F	del

Appendix C

Extensions to Standard Pascal

This appendix describes DOMAIN Pascal's extensions to ISO standard Pascal.

C.1 Extensions to Program Organization

Chapter 2 describes the elements that make up a Pascal program. This section describes the extensions to standard Pascal.

C.1.1 Identifiers

Although an identifier must begin with a letter, you can include underscores (_) or dollar signs ($) in the name. For example, `mailing_$lists` is a legal identifier.

C.1.2 Integers

You can specify integers in any base from 2 to 16. To do so, use the following syntax:

base#value

For **base**, enter an integer from 2 to 16. For **value** enter any integer within that base. If the **base** is greater than 10, use the letters A through F (or a through f) to represent digits with the values 10 through 15.

For example, consider the following integer constant declarations:

```
half_life  := 5260;     /* default     (base 10) */
hexograms  := 16#1c6;   /* hexadecimal (base 16) */
luck       := 2#10010;  /* binary      (base 2)  */
wheat      := 8#723;    /* octal       (base 8)  */
```

C.1.3 Comments

You can specify comments in any of the following three ways:

```
{ comment }
(* comment *)
"comment"
```

(The spaces before and after the comment delimiters are for clarity only; you don't have to include these spaces.) For example, here are three comments:

```
{ This is a comment. }
(* This is a comment. *)
"This is a comment."
```

Unlike standard Pascal, the comment delimiters of DOMAIN Pascal must match. For example, a comment that starts with a left brace doesn't end until the compiler encounters a right brace. Therefore, you can nest comments, for example:

```
{ You can (*nest*) comments inside other comments. }
```

The DOMAIN Pascal compiler ignores the text of the comment, and interprets the first matching delimiter as the end of the comment.

Standard Pascal does not permit nested comments. If you want to use unmatched comment delimiters, as standard Pascal allows, you must compile with the **–iso** switch. Chapter 6 describes that switch.

Finally, DOMAIN Pascal permits you to put compiler directives inside comment delimiters. However, if you do so, you *cannot* use spaces; see the listing for "Compiler Directives" in Chapter 4 for details.

C.1.4 Sections

DOMAIN Pascal allows you to assign code and data in your program to a nondefault section. A section is a named contiguous area of main memory.

C.1.5 Declarations

You can declare the **label**, **const**, **type**, and **var** declaration parts in any order. You can specify the declaration parts an unlimited number of times.

In addition to **label**, **const**, **type**, and **var** declaration parts, you can also declare a **define** part (which is detailed in Chapter 7).

C.1.6 Constants

You can set a constant equal to a real, integer, string, char, or set expression. The constant can also be the pointer expression **nil**. The expression can contain the following types of terms:

- A real number, an integer, a character, a string, a set, a Boolean, or **nil**
- A constant that has already been defined in the **const** declaration part (note that you cannot use a variable here)
- Any predefined DOMAIN Pascal function (e.g., **chr**, **sqr**, **lshft**, **sizeof**), but only if the argument to the function is a constant
- A type transfer function

You can optionally separate these terms with any of the following operators:

Operator	**Data Type of Operand**
+, –, *	Integer, real, or set
/	Real
mod, div, !, &, ~	Integer

For example, the following **const** declaration part defines eight constants:

```
CONST
   x = 10;
   y = 100;
   z = x + y;
   current_year = 1994;
   leap_offset  = (current_year mod 4);
   bell         = chr(7);
   pathname     = '//et/go_home';
   pathname_len = sizeof(pathname);
```

C.1.7 Labels

In standard Pascal, only integers can be used as labels. In DOMAIN Pascal, you can use both identifiers and integers as labels.

C.2 Extensions to Data Types

Chapter 3 describes the data types supported by DOMAIN Pascal. This section describes the extensions that DOMAIN Pascal supports.

C.2.1 Initializing Variables in the Var Declaration Part

DOMAIN Pascal lets you initialize variables in the **var** declaration part. You can initialize integer, real, Boolean, char, subrange, set, enumerated, array, record, and pointer variables with an assignment statement following the data type; for example:

```
VAR
    x : integer := 17;
    r : real := 5.3E-14;
    a : array[1..7] of char := 'Wyoming';
```

For arrays in particular, DOMAIN Pascal supports many extensions for simplifying initialization. See Chapter 3 for details.

C.2.2 Integers

DOMAIN Pascal supports the following two nonstandard predeclared integer data types:

- **Integer16** -- Use it to declare a signed 16-bit integer. (**Integer** and **integer16** have identical meanings.)
- **Integer32** -- Use it to declare a signed 32-bit integer. A signed 32-bit integer variable can be any value from –2147483648 to +2147483647.

C.2.3 Reals

DOMAIN Pascal supports the following two nonstandard predeclared real data types:

- **Single** -- Same as **real.**
- **Double** -- Use it to declare a signed double-precision real variable. DOMAIN Pascal represents a double-precision real number in 64-bits. A double-precision real variable has approximately 16 significant digits.

C.2.4 Pointer Types

In addition to the standard pointer type, DOMAIN Pascal supports a **univ_ptr** type and a special pointer type that points to procedures and functions.

The predeclared data type **univ_ptr** is a universal pointer type. A variable of type **univ_ptr** can point to a variable of any type. You can use a **univ_ptr** variable in the following contexts only:

- Comparison with a pointer of any type
- Assignment to or from a pointer of any type
- Formal or actual parameter for any pointer type
- Assignment to the result of a function

DOMAIN Pascal supports a special pointer data type that points to a procedure or a function. By using procedure and function data types, you can pass the addresses of routines obtained with the **addr** predeclared function. (See the **addr** listing of Chapter 4 for a description of this function.) You may only obtain the addresses of top-level procedures and functions; you cannot obtain the addresses of nested or explicitly declared **internal** procedures and functions. (See Chapter 5 for details about **internal** procedures.)

C.2.5 Named Sections

By default, DOMAIN Pascal stores all variables declared in the **var** declaration part at the program or module level to the DATA$ section. However, DOMAIN Pascal enables you to assign variables to sections other than DATA$. Named variable sections are synonymous with named common blocks in FORTRAN.

C.2.6 Variable and Type Attributes

DOMAIN Pascal supports attributes for variables and types. These attributes supply additional information to the compiler when you declare a variable or a type.

DOMAIN Pascal currently supports three of these attributes: **volatile, device,** and **address**. The **volatile** and **device** attributes enable you to turn off certain optimizations that would otherwise ruin programs that access device registers or shared memory locations. The **address** attribute associates a variable with a specific virtual address.

C.3 Extensions to Code

Chapter 4 describes the action part (the executable block) of a Pascal program. This section describes the extensions.

C.3.1 Bit Operators

DOMAIN Pascal supports three bit operators for bitwise *and*, *not*, and *or* operations. These operators perform Boolean operations by comparing the bits in each bit position of two integer arguments. For details on these three operators see the "Bit Operators" listing in Chapter 4. In addition to these three bit operators, DOMAIN Pascal supports an **xor** function which performs a bitwise exclusive or operation on two integers.

C.3.2 Bit-Shift Functions

DOMAIN Pascal supports the following three bit-shift functions:

- **Rshft**, which shifts the bits in an integer a specified number of spaces to the right.
- **Arshft**, which shifts the bits in an integer a specified number of spaces to the right and preserves the sign of the integer.
- **Lshft**, which shifts the bits in an integer a specified number of spaces to the left.

For syntax details, see the **rshft**, **arshft**, and **lshft** listings in Chapter 4.

C.3.3 Compiler Directives

DOMAIN Pascal supports the compiler directives shown in Table 4-9 in Chapter 4.

You can place a directive anywhere in your program. To use a directive, specify its name in a comment or as a statement. For example, all of the following formats are valid:

```
{%directive}
(*%directive*)
%directive
```

If you specify a directive within a comment, the percent sign must be the first character after the delimiter. (Spaces count as characters.)

C.3.4 Addr Function

DOMAIN Pascal supports an **addr** function that returns the virtual address of the specified variable or routine. For syntax details, see the **addr** listing of Chapter 4.

C.3.5 Max and Min Functions

DOMAIN Pascal supports a **max** and a **min** function for finding the larger and smaller of two operands, respectively.

C.3.6 Discard Procedure

DOMAIN Pascal supports a **discard** procedure for explicitly discarding the computed value of an expression. It usually is used with a function (which in standard Pascal *must* return a value) for which the value is not needed. The optimizer may eliminate the computation and issue a warning message if the return value isn't used, but **discard** explicitly throws away the computed value and so eliminates the warning message.

C.3.7 I/O Procedures

DOMAIN Pascal supports the I/O procedures of standard Pascal, plus the following four procedures:

- **Open**, which opens permanent files for I/O access.
- **Close**, which explicitly closes an open file.
- **Find**, which locates a specific element in a record–structured file.
- **Replace**, which modifies an existing element in a record–structured file.

As in standard Pascal, you can create a temporary file with the **rewrite** procedure; however, by using the **open** procedure, you can create a permanent file. (Here, "permanent" means a file that exists even after the program terminates.)

When a program terminates, the operating system automatically closes any open files. However, because an open file can clog system resources, DOMAIN Pascal provides a **close** procedure that allows you to close a file from within your program.

The **find** procedure locates records from a record–structured file. Records here refer to the elements in a file whose file variable was declared as a **file of** data type. By using **replace** in combination with **find**, you can replace an existing record.

From a DOMAIN Pascal program, you can easily access Input Output Stream calls (known as IOS calls) and formatting calls (known as VFMT calls).

For syntax details, see the **open**, **close**, **find**, and **replace** listings in Chapter 4. For an overview of I/O (including IOS calls and VFMT calls), see Chapter 8.

C.3.8 If Statement

DOMAIN Pascal supports an **and then** and **or else** extension to the **if** statement. You can use **and then** and **or else** to guarantee that DOMAIN Pascal will evaluate the Boolean expressions of a condition in the order that you write them. They also guarantee "short–circuit" evaluation; that is, at runtime, the system will only evaluate as many expressions as is necessary. For details, see the **if** listing in Chapter 4.

C.3.9 Loops

DOMAIN Pascal supports **for, while,** and **repeat,** which are the three looping statements of standard Pascal. DOMAIN Pascal also supplies the following two additional statements for further control within a loop:

- A **next** statement for skipping over the current iteration of a loop
- An **exit** statement for unconditionally jumping out of the current loop

For syntax details, see the **next** and **exit** listings of Chapter 4.

C.3.10 Range of a Specified Data Type

DOMAIN Pascal supports

- A **firstof** function for returning the first possible value of a specified data type
- A **lastof** function for returning the last possible value of a specified data type

For syntax details, see the **firstof** and **lastof** listings in Chapter 4.

C.3.11 Integer Subrange Testing

By default, DOMAIN Pascal does not check the value of input data to see that it falls within the defined range of a variable declared as a subrange of integers. To ensure subrange checking, call the **in_range** function. For syntax details, see the **in_range** listing in Chapter 4.

C.3.12 Extensions to Read and Readln

In addition to allowing input into any real, integer, char, or subrange variable, as standard Pascal allows, DOMAIN Pascal's **read** and **readln** also allow input to a Boolean or enumerated variable.

C.3.13 Premature Return From Routines

As in standard Pascal, DOMAIN Pascal returns control to the calling routine after executing the last line in the called routine. If you want to return to the calling routine before reaching the last line, you can issue a **return** statement. For syntax details, see the **return** listing in Chapter 4.

C.3.14 Memory Allocation of a Variable

DOMAIN Pascal supports a **sizeof** function. This function returns the size (in bytes) that a data type (predeclared or user-defined), variable, constant, or string inhabits in main memory.

C.3.15 Extensions to With

DOMAIN Pascal supports the standard format of **with** and also supports the following alternative format:

with v1:identifier1, *v2:identifier2, ... vN:identifierN* **do**
    stmnt;

An identifier is a pseudonym for the record variable v. To specify a record, use the identifier instead of the record variable v. Furthermore, to specify a field in a record, use identifier.field_name rather than v.field_name.

For example, given the following record declaration

```
basketball_team = record
    mascot    : array[1..15] of char;
    height    : single;
end;
```

consider the following three methods of assigning values:

```
readln(basketball_team.mascot);     {Not using WITH.}
readln(basketball_team.height);     {Not using WITH.}

WITH basketball_team DO        {Using standard WITH.}
   begin
       readln(mascot);
       readln(height);
   end;


WITH basketball_team : B  DO  {Using extension to WITH.}
   begin
       readln(B.mascot);
       readln(B.height);
   end;
```

The extension is useful for working with long record names when two records contain fields that have the same names.

C.3.16 Type Transfer Functions

DOMAIN Pascal supports type transfer functions which enable you to change the type of a variable or expression within a statement. To perform a type transfer function, use any user-created or standard type name as if it were a function name in order to "map" the value of its argument into that type.

With one exception, the size of the argument must be the same as the size of the destination type. (Chapter 3 describes the size of each data type). This size equality is required because the type transfer function does not change any bits in the argument. DOMAIN Pascal just "sees" the argument as a value of the new type. The one exception is that integer subranges are compatible.

C.3.17 Extensions to Write and Writeln

DOMAIN Pascal allows you to specify a negative field width for **chars**, **strings**, and arrays of **chars**. Also, if you specify a one-part field width for a real number, DOMAIN Pascal adds or removes leading blanks. See the listing for **write** and **writeln** in Chapter 4 for details.

C.4 Extensions to Routines

Chapter 5 describes procedures and functions. The term "routine" means either procedure or function. Also, the term "argument" refers to the data passed to a routine while "parameter" means the templates for the data to be received.

The following subsections describe DOMAIN Pascal's extensions to routine calling.

C.4.1 Direction of Data Transfer

In standard Pascal, you cannot specify the direction of parameter passing. However, DOMAIN Pascal supports extensions to overcome this problem. You can use the following keywords in your routine declaration:

- **In** -- This keyword tells the compiler that you are going to pass a value to this parameter, and that the routine is not allowed to alter its value. If the called routine does attempt to change its value (i.e., use it on the left side of an assignment statement), the compiler issues an "Assignment to IN argument" error.

- **Out** -- This keyword tells the compiler that you are not going to pass a value to the parameter, but that you expect the routine to assign a value to the parameter. It is incorrect to try to use the parameter before the routine has assigned a value to it, although the compiler does not issue a warning or error in this case.

  If the called routine does not attempt to assign a value to the parameter, the compiler may issue a "Variable was not initialized before this use" warning. This could occur if your routine only assigns a value to the parameter under certain conditions. If that is the case, you should designate the parameter as **var** instead of **out**.

  In some cases, the compiler cannot determine whether all paths leading to an **out** parameter assign a value to it. If that happens, the compiler does not issue a warning message.

- **In out** -- This keyword tells the compiler that you are going to pass a value to the parameter, and that the called routine is permitted to modify this value. It is incorrect to call the routine before assigning a value to the parameter, although the compiler does not issue a warning or error in this case. The compiler also doesn't complain if the called routine does not attempt to modify this value.

C.5 Universal Parameter Specification

By default, DOMAIN Pascal and standard Pascal check to ensure that the argument you pass to a routine has the same data type as the parameter you defined for the routine. As an extension, you can tell DOMAIN Pascal to suppress this type checking. You do this by using the keyword **univ** prior to a type name in a parameter list. By using **univ**, you can pass an argument that has a different data type than its corresponding parameter.

Univ is especially useful for passing arrays.

C.5.1 Routine Options

Standard Pascal supports a **forward** option. DOMAIN Pascal supports the **forward** option, and also supports the following routine options:

- **Extern** -- By default, Pascal expects a called routine to be defined within the source code file where it is called. The **extern** option tells the compiler that the routine is possibly defined outside of this source code file.
- **Internal** -- By default, all routines defined in modules become global symbols. But, if you declare the routine with the **internal** option, the compiler makes the routine a local symbol.
- **Variable** -- By default, you must pass the same number of arguments to a routine each time you call the routine. However, by using the **variable** option in a routine declaration, you can pass a variable number of arguments to the routine.
- **Abnormal** -- This option warns the compiler that a routine can cause an abnormal transfer of control.
- **Val_param** -- By default, DOMAIN Pascal passes arguments by reference. However, by using the **val_param** option, you tell DOMAIN Pascal to pass arguments by value.
- **Nosave** -- This option indicates that the contents of data registers D2 through D7, address registers A2 through A4, and floating-point registers FP2 through FP7 will not be saved when a called assembly language routine finishes and returns to the DOMAIN Pascal program. However, two registers are preserved: A5, which holds the pointer to the current stack area, and A6, which holds the address of the current stack frame.
- **Noreturn** -- This option specifies an unconditional transfer of control; once a routine marked **noreturn** is called, control can never return to the caller.
- **D0_return** -- By default, a Pascal function returning the value of a pointer type variable puts that value in address register A0. **D0_return** tells the compiler to put the value in A0 *and* data register D0.

C.5.2 Routine Attribute List

You can specify a routine attribute list when you declare a routine. Within the routine attribute list, you can specify a nondefault section name.

A "section" is a named contiguous area of an executing object. (Refer to the *DOMAIN Binder and Librarian Reference* for full details on sections.) By default, the compiler assigns code to the PROCEDURE$ section and data to the DATA$ section. Thus, by default, all code from every routine in the program is assigned to PROCEDURE$, and all data from every routine in the program is assigned to DATA$. However, DOMAIN Pascal permits you to override the default of PROCEDURE$ and DATA$ on a routine-by-routine basis. (You can also override the defaults on a variable-by-variable or module-by-module basis.) This makes it possible to organize the runtime placement of routines so that logically related routines can share the same page of main memory and thus reduce page faults. Conversely, you can declare a rarely called routine as being in a separate section from the frequently called routines.

C.6 Modularity

DOMAIN Pascal allows you to break your program into separately compiled source files. After compiling all the source files, you can bind the resulting objects into one executable object file. Chapters 6 and 7 document the details.

C.7 Other Features of DOMAIN Pascal

DOMAIN Pascal supports many other features, such as the ability to call routines written in other DOMAIN languages. However, the remaining features are all implementation–dependent features, and not actual extensions.

Appendix D

Deviations From Standard Pascal

This appendix describes DOMAIN Pascal's deviations from ISO standard Pascal, and documents the sections in the ISO standard document to which DOMAIN Pascal does not completely adhere.

D.1 Deviations From the Standard

DOMAIN Pascal does not include certain features of standard Pascal, and this list documents the deviations:

- Identifiers in standard Pascal may be longer than 32 characters and still be considered unique.
- DOMAIN Pascal ignores the file list in the program heading.
- DOMAIN Pascal allows arrays of up to seven dimensions only.
- Although DOMAIN Pascal recognizes the **packed** syntax, it does not support **packed** arrays or **packed** sets.

D.2 Deviations From Specific Sections of the Standard

The following lists the sections in the ISO standard document to which DOMAIN Pascal does not completely adhere, and the reason it does not adhere to that section.

6.1.2	You may redeclare NIL.
6.1.5	You are not required to include a sequence of digits after a period in a floating-point number.

6.1.8 You are not required to leave a blank between a number and a word-type operator. For example, DOMAIN Pascal accepts the following:

```
result := 10mod 1;
```

6.2.2 The following type of declaration works under DOMAIN Pascal:

```
TYPE
    rec = record
          ptr    : ^my_var;
          my_var : integer
    end;
    my_var = rec;
```

6.4.3.3 There is no requirement that all values of a tag-type in a record appear as **case** constants.

DOMAIN Pascal does not detect a reference to an inactive variant field. Also, it does not mark variant fields as inactive when a new variant tag becomes active.

6.4.5 Subranges of the same type that are defined with different ranges are considered identical.

DOMAIN Pascal considers structurally identical types to be identical. For example, the following are identical under DOMAIN Pascal:

```
TYPE
    first = array[1..20] of integer32;
    second = array[1..20] of integer32;
```

Also, DOMAIN Pascal ignores the keyword **packed**, so structurally identical sets are considered identical.

6.4.6 You may assign structured types containing a file component to each other.

You may make an assignment from an integer expression that includes a value outside one of the variables' declared subranges. Also, you may pass an integer argument that is outside the corresponding parameter's declared subrange. In addition, DOMAIN Pascal considers sets of different subranges from the same enumerated type to be compatible for assignment and as parameters.

6.5.5 DOMAIN Pascal does not detect when a field variable passed as a **var** parameter is modified. It also does not detect a modification to a file variable when a reference to the buffer exists.

6.6.3.3 You may pass the selector of a variant or a component of a **packed** variable as a **var** parameter. Also, DOMAIN Pascal accepts a procedure call like the following where x is declared in the procedure heading as being a **var** parameter:

```
proc_name((x));
```

6.6.3.5, 6.6.3.6 DOMAIN Pascal treats **integer** and subrange of **integer** as identical.

6.6.3.6 DOMAIN Pascal considers the following to be identical:

```
VAR                         VAR
    a : integer;                a,b : integer;
    b : integer;
```

6.6.5.2 DOMAIN Pascal does not detect a **put** of an undefined buffer variable at compiletime. It does not consider a **read** of an enumerated type to be an error, or an assignment from a file variable of an enumerated type followed by a **get** to be an error. You may write an integer expression that includes a value outside one of the variables' declared subranges. Also, you may make an assignment from an integer expression that includes a value outside one of the file variables' declared subranges.

6.6.5.3 You may **dispose** a pointer that is active because it has been de–referenced as a parameter or in a **with** block. In addition, DOMAIN Pascal does not report an error if you use a pointer variable after you have **disposed** of it or if you **dispose** of a dangling pointer (that is, a pointer with an address assigned to another pointer).

You also may use a record allocated with a long form of **new** as an operand in an expression or as a variable in an assignment statement. You may pass as an argument a variable that was allocated with **new** and that uses variant tags.

DOMAIN Pascal does not report an error if you use different tags for a variable in **new** and **dispose.** It also does not detect the activation of a variant on a variable that **new** allocated with a different tag, or if your program includes illegal variant tags in a **dispose.**

6.6.5.4 The **pack** procedure accepts a normal array where **packed** is expected, and a **packed** array where a normal array is expected.

In **pack,** DOMAIN Pascal does not detect an uninitialized component in the unpacked array. Similarly, in **unpack,** DOMAIN Pascal does not detect an uninitialized component in the packed array.

6.6.6.2 On some nodes, the **sqr** function does not detect overflow.

6.6.6.3 On some nodes, **trunc** and **round** do not detect overflow.

6.6.6.4 **Succ, pred,** and **chr** do not detect overflow.

6.7.2.2 DOMAIN Pascal does not detect an overflow or underflow on integer arithmetic. Also, it allows you to supply a negative value for j in an expression like the following:

```
i mod j
```

6.7.2.4 DOMAIN Pascal does not detect operations on overlapping sets with incompatible elements.

6.8.1 DOMAIN Pascal permits jumps between branches of a **case** statement and from one structured statement into the middle of another.

6.8.3.5 DOMAIN Pascal does not detect the lack of a **case** statement constant corresponding to a runtime **case** value.

6.8.3.9 DOMAIN Pascal does not detect an underflow of an assignment from **pred** to a **for** statement index variable. Also, it does not issue an error if there is an overflow in the final value of a **for** statement index variable.

DOMAIN Pascal does not detect the possibility that an inner block will change the value of a **for** statement's index variable. Also, DOMAIN Pascal allows a non–local variable, a formal parameter, or a value parameter to be used as a **for** statement's index variable. You also can use a program level global variable as the index variable for a **for** statement that resides in an inner block.

6.9.1	DOMAIN Pascal does not detect an overflow of a subrange boundary for a **read** statement.
6.9.3.1	You may supply a nonpositive field width or a nonpositive fractional–digits field width to a **write** or **writeln.**
6.10	You don't have to declare **input** and **output** in a **program** heading to use them in the program. You can, however, repeat parameters in a **program** heading (e.g., `program testing (output, output);`) or redeclare program parameters as some type other than a file. Also, you don't have to declare program parameters in the **var** declaration part of your program.

Appendix E

Systems Programming Routines

DOMAIN Pascal includes several routines designed specifically for systems programmers' use. Systems programmers are those who need to do very low–level work in their programs and who need direct access to specific registers and bits within those registers. They frequently write some programs in Pascal and some in assembly language.

If you are a systems programmer, you might use these routines when writing device drivers, or when doing other low–level manipulations of the hardware status register.

E.1 Overview

Table E–1 briefly describes the available systems programming routines, all of which are extensions to standard Pascal. A more complete explanation follows.

Table E–1. Systems Programming Routines

Routine	Action
disable	Turns off the interrupt enable in the hardware status register.
enable	Turns on the interrupt enable in the hardware status register.
set_sr	Saves the current value of the hardware status register and then inserts a new value.

E.2 Restrictions for Use

All the routines described in this appendix generate privileged instructions and may only be executed from supervisor mode. If you try to run a program using one of these routines while in user mode, you get a privilege-violation error.

Disable -- Turns off the interrupt enable in the hardware status register. (Extension)

FORMAT

disable {**disable** is a procedure.}

Argument

Disable takes no arguments.

DESCRIPTION

Disable is a built–in procedure for systems programmers' use. It turns off the interrupt enable in the hardware status register and should be used with its complementary procedure **enable**.

By turning off the interrupt enable, **disable** allows you to prevent an interrupt from coming in while the program is in a critical section. After the critical section finishes, you should use **enable** to turn the interrupt enable back on.

The **disable–enable** pair look like this in code:

```
disable;

   { Critical section.                                }
   { No interrupts allowed while this section is executing. }

enable;
```

If you mistakenly use only **disable**, your program will essentially grind to a halt since no interrupt signals will be able to get to it. You should only use the **disable–enable** pair around very small sections of code.

Enable -- Turns on the interrupt enable in the hardware status register. (Extension)

FORMAT

enable {**enable** is a procedure.}

Argument

Enable takes no arguments.

DESCRIPTION

Enable is a built-in procedure for systems programmers' use. It turns on the interrupt enable in the hardware status register and usually is used with its complementary procedure **disable**.

By turning on the interrupt enable, **enable** allows your program to receive interrupts. Usually, **disable** will have been used to prevent the reception of interrupts during a critical section of code. After the critical section finishes, **enable** lets the interrupts flow.

The **disable-enable** pair look like this in code:

```
disable;

   { Critical section.                               }
   { No interrupts allowed while this section is executing. }

enable;
```

Because the interrupt enable is turned on by default, there is no effect if you mistakenly use only **enable** in your program.

Set_sr -- Saves the current value of the hardware status register and then inserts a new one. (Extension)

FORMAT

oldsr := **set_sr**(newsr);   {**set_sr** is a function.}

Argument

oldsr	The old value of the hardware status register.
newsr	The new value of the hardware status register.

DESCRIPTION

Set_sr is a built-in function for systems programmers' use. It reads the hardware status register (SR) and replaces its current value with newsr. The original value then is assigned to oldsr. This translates to assembly language code something like this:

```
move.w    SR,d0
move.w    newsr,SR
move.w    d0,oldsr
```

The function eliminates six instructions in what often was a time-critical path.

Index

The letter *f* means "and the following page"; the letters *ff* mean "and the following pages". Symbols are listed at the beginning of the index.

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *DOMAIN Pascal Language Reference*
Order No.: 000792 Revision: 04 Date of Publication: January, 1987

What type of user are you?

_____ System programmer; language ____________	
_____ Applications programmer; language ____________	
_____ System maintenance person	_____ Manager/Professional
_____ System Administrator	_____ Technical Professional
_____ Student Programmer	_____ Novice
_____ Other	

How often do you use the DOMAIN system?____________________

What parts of the manual are especially useful for the job you are doing?____________________

What additional information would you like the manual to include?____________________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)____________________

Your Name Date

Organization

Street Address

City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD