PASCAL

IDR4303

Revision Ø

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 17 (Rev. 17).

PRIME Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts Ø1701

ACKNOWLEDGEMENTS

We wish to thank the members of the documentation team and also the non-team members, both customer and Prime, who contributed to and reviewed this book.

First Printing October 1980

All correspondence on suggested changes to this document should be directed to:

Grace T. Na
Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

CONTENTS

PART I - INTRODUCTION

PART II - LANGUAGE-SPECIFIC SYSTEM INFORMATION

PART III - LANGUAGE REFERENCE

SECTION 1

INTRODUCTION

This document is a programmer's reference guide to the Pascal language as implemented on the Prime system.

You are expected to be familiar with the Pascal language, and with programming in general, but not necessarily with Prime computers. If you are unfamiliar with the language read one of the many commercially available instruction books, such as:

Jensen, Kathleen and Wirth, Niklaus, PASCAL User Manual And Report, Second Edition. Springer-Verlag, New York, 1978.

Schneider, G., Weingart, S. and Perlman, D., An Introduction To Programming And Problem Solving With PASCAL, John Wiley & Sons, Inc., New York, 1978.

CONTENTS OF THIS GUIDE

This document contains the following:

- An overview of the Pascal language as implemented on Prime computers (Section 1)
- Complete information on the use of the Pascal compiler (Section 2)
- An introduction to loading and executing Pascal programs (Section 3)
- A reference for the Pascal language based upon the draft proposal "ISO/DP7185", including complete information on all Prime extensions and restrictions to the language (Sections 4-11)
- A list of Pascal compile-time and run-time error messages and their meanings (Appendix A)
- Storage formats used for the Pascal data types (Appendix B) and the ASCII character set (Appendix C)

RELATED DOCUMENTS

The following documents contain additional information relevant to programming in Pascal.

The Prime User's Guide

Complete instructions for creating, loading and executing programs in Prime Pascal or any Prime language, plus extensive additional information on Prime system utilities for programmers, are found in The Prime User's Guide. The Prime User's Guide and The Pascal Reference Guide are both essential to the Pascal programmer.

The user's guide also contains a complete guide to all Prime documentation.

The Draft Proposal "ISO/DP7185" Programming Language Pascal

The definitive reference for Pascal is The Draft Proposal "ISO/DP7185" Programming Language Pascal. Every installation which uses Pascal extensively should have a copy of this proposed standard, which may be obtained from American National Standards Institute, 1430 Broadway, New York, NY 10018.

The New User's Guide to EDITOR and RUNOFF

Prime's EDITOR is an interactive text-editing utility. It is used to enter and modify text in the computer. New programs that do not rely on cards or tapes are usually input to the system at a terminal using EDITOR.

The New User's Guide to EDITOR and RUNOFF contains a complete description of the EDITOR. It also provides a basic introduction to the Prime system for those with little or no computer experience, and describes RUNOFF, Prime's text-formatting utility.

The LOAD and SEG Reference Guide

Ordinarily, to load and execute programs you need only the information given in The Pascal Reference Guide or The Prime User's Guide. If you wish to control the load process in more detail, or use the full range of Prime loader capabilities, see The LOAD and SEG Reference Guide.

The PRIMOS Subroutines Reference Guide

Prime offers a large selection of applications-level subroutines and PRIMOS operating system subroutines which can be declared as external in procedure/function declarations of a Pascal program, then referenced from any point within the program. These routines are described in The

PRIMOS Subroutines Reference Guide. (See also Section 9 of this guide.)

THE PASCAL LANGUAGE

Pascal is a multi-purpose structured language that can be used for system, commercial and scientific data processing. Pascal is also used as the principal instructional language in many educational institutions. This language, developed in 1968 by Professor Niklaus Wirth at the Eidgenossische Technische Hochschule (ETH) in Zurich, Switzerland, is a descendent of the language ALGOL-60. Pascal is named for the French mathematician Blaise Pascal.

PRIME EXTENSIONS TO PASCAL

Prime Pascal is based upon The Draft Proposal "ISO/DP7185" Programming Language Pascal with the extensions listed below. They are discussed in detail in the sections noted.

- The dollar sign $ and underscore _ may be used in identifiers. (See Section 4.)
- Text from an external file can be included during compilation by using the compiler directive %INCLUDE. (See Section 5.)
- The infix operators & and ! perform the bitwise AND and OR operations for type INTEGER operands. (See Section 7.)
- The CASE statement has been extended to include the optional OTHERWISE clause. (See Section 8.)
- A procedure or function can be declared as external by appending the directive EXTERN to its procedure or function declaration, respectively. (See Section 9.)
- Separate compilation of Pascal procedures and/or functions can be accomplished by including the {$E+} compiler switch at the beginning of the module. (See Sections 2 and 9.)
- An optional second argument has been added to the standard procedures RESET and REWRITE to denote the name of the file to be opened for input or output, respectively. (See Section 10.)
- A procedure CLOSE has been added which takes the variable of the file to be closed as its argument. Files must be closed explicitly using this procedure, otherwise they remain open after program termination. (See Section 10.)
- Predefined file-variables INPUT and OUTPUT provide I/O to and from the user's terminal. (See Section 6.)

- The program heading to a Pascal program is optional. (See Section 5.)

PRIME RESTRICTIONS TO PASCAL

Restrictions to the Proposed Standard

- An identifier may contain up to 32 significant characters in its spelling. Identifiers greater than 32 characters in length will result in a severity 3 error at compile time. (See Section 4.)

- The standard procedures - READ, READLN, WRITE, and WRITELN - are only applicable to files of the type FILE OF CHAR. The standard procedures GET and PUT should be used to do all I/O on files of any other type. (See Section 10.)

- Only value and variable parameters are supported in a procedure or function declaration. Procedure and function parameters are not currently supported. (See Section 9.)

- The keyword PACKED used in type definitions does not have any effect. However, use of PACKED is not advised, and will generate a severity 1 error (warning) at compilation. The standard procedures PACK and UNPACK will also generate a warning, but the correct code will be generated. (See Sections 6 and 9.)

Implementation Characteristics

- Prime's character set (the ANSI, ASCII, 7-bit character set plus 1 parity bit) has ordinal values between octal 200 and 377. (See Appendix C.) Examples:

```
CHR(161) = !
CHR(33) = !        {CHR(33) prints as ! on the terminal but
                    it is not generally compatible with Prime's
                    character set.}
```

- String literals are restricted in length to 256 characters. (See Section 6.)

- All SET types are restricted to 256 elements. (See Section 6.)

- The value of MAXINT is 32767. (See Section 6.)

- The approximate range of real numbers is $-1*10^{38}$ to $+1*10^{38}$. (See Section 6.)

INTERFACE TO OTHER LANGUAGES

Since all Prime high-level languages are alike at the object-code level, and since all use the same calling conventions, object modules produced by the Pascal compiler can reference and be referenced by modules produced by the F77, FTN, COBOL, or PL1G compilers, provided that certain restrictions are observed:

- All I/O routines must be written in the same language.
- There must be no conflict of data types for variables being passed as arguments. For example, an INTEGER in Pascal should be declared as FIXED BINARY (15) in PL/I. See Appendix B for a description of Pascal data storage formats.
- Modules compiled in 64V or 32I mode cannot reference or be referenced by modules compiled in any R mode. Modules in 64V or 32I may reference each other if they are otherwise compatible.

Pascal program units can also reference PMA (Prime Macro Assembler) routines, and vice versa. For information, see The Assembly Language Programmer's Guide.

CONVENTIONS USED IN THIS GUIDE

Various conventions are used in the following sections. Their meanings must be clearly understood by the reader.

Conventions Indicating Extensions

Every Pascal extension is labeled as such in the text of this guide. When a specific feature is explicitly described as being a Pascal extension, the feature should not be used in a program which may have to run on a non-Prime system.

When the Pascal language is mentioned in general, the reference is to Prime's Pascal as a whole.

Conventions in Examples

In all examples involving dialog between the user and the system, the user's input is underlined, and the system's output is not. For example:

```
OK, attach mydirec
OK, ed oldfile
GO
EDIT
```

Examples consisting only of Pascal statements, with no responses from

the system, are not underlined. For example:

```
BEGIN
  I := 'SAMPLE';
  WRITELN (I)
END
```

Typographical Conventions

WORDS-IN-UPPERCASE	Uppercase letters identify command words or standard identifiers, and underlined uppercase letters identify keywords. They are to be entered literally.
words-in-lowercase	Lowercase letters identify options or arguments (objects). The user substitutes an appropriate numerical or text value.
Brackets [ ]	Brackets indicate that the item enclosed is optional.
Vertical Slash |	Vertical slashes separate alternative options in an options list. Unless the list of options is enclosed by brackets, one option must be selected.
Parentheses ( )	When parentheses appear in a statement format, they must be included literally when the statement is used.
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.

SECTION 2

USING THE PASCAL COMPILER

INTRODUCTION

Prime's Pascal compiler accepts a source program meeting the requirements of Prime's Pascal as specified in this manual. It can output an object file, a source listing, error and statistics information, and various messages. Errors are printed at the terminal as the compiler detects them.

This section tells:

- How to invoke the compiler
- How to specify options to the compiler
- The significances of the various messages that are printed during compilation
- The meanings of the various compiler options
- How to specify switches to the compiler
- The meanings of the various compiler switches

INVOKING THE COMPILER

The Pascal Compiler is invoked by the PASCAL command to PRIMOS:

```
PASCAL pathname [-option]...
```

pathname The pathname of the Pascal source program to be compiled.

options Mnemonics for the options controlling compiler functions.

All mnemonic options begin with a hyphen "-". Example:

```
PASCAL TEST1 -RANGE -LISTING
```

will cause TEST1 to be compiled with the options given.

COMPILER ERROR MESSAGES

For each error encountered in the program, an error message will be automatically printed at the terminal and in the source listing if one exists. The general format of an error message is:

```
ERROR xxx SEVERITY y explanation
```

xxx — Error Code

y — Severity code

explanation — Description of the error, and possible remedies.

The significance of the severity code is:

Severity	Description
1	Warning
2	Error that has been corrected
3	Uncorrected error - prevents optimization and code generation
4	Error that prevents further compilation

Example:

```
ERROR 56 SEVERITY 3  Missing ";"
```

A complete list of Pascal compiler error messages is provided in Appendix A.

END-OF-COMPILATION MESSAGE

After the compilation is complete, the compiler prints an end-of-compilation message at the terminal. Its format is:

```
xxxx ERRORS (PASCAL-REV zz.z)
```

xxxx — The number of compilation errors (0000 indicates a successful compilation).

zz.z — The current revision number of the Pascal compiler.

Example:

```
0001 ERRORS (PASCAL-REV 1.4)
```

After compilation, control returns to PRIMOS.

COMPILER OPTIONS

The available compiler options can be categorized as follows:

- Specify the source file
- Specify the existence and properties of the object code
- Specify the existence and contents of the source listing
- Specify the handling of error and statistics information

Compiler options generally come in pairs: for each option, there is an option having the opposite effect. Most option pairs direct the compiler to do or not do some action. A few present a choice between two actions. One member of each pair is always the default.

Not all options can be specified explicitly. When either member of an option pair could be a desirable default at some installation, both options are explicitly available, so that the default can always be specified by the System Administrator. When only one member is a desirable default, that option cannot be explicitly specified; it is selected by simply accepting the default.

In the following list, each option is given with its opposite. Options which cannot be given explicitly are printed in lowercase without an initial hyphen. For each pair, the Prime-supplied default is underlined. Commonly used options are marked with an asterisk; new users should skip over options without asterisks.

Some options require an object in addition to the option specification. The object follows the option, and is not preceded by a hyphen. Options may be given in any order.

Table 2-1 lists the options in the order that they are discussed below.

Table 2-1. Compiler Options

Specify the Source File

-S or -I	Gives name of source file

Specify the Existence and Properties of the Object Code

*	-B [object]	Controls existence of object file
	-BIG / nobig	Controls boundary-spanning code
	-64V / -32I	Controls addressing mode
*	-DEBUG / nodebug	Generates debugger code
*	-OPTIMIZE / -NOOPTIMIZE	Controls optimization
	-PRODUCTION / noproduction	Generates debugger code
*	-RANGE / -NORANGE	Controls insertion of range-checking code
	-EXTERNAL / noexternal	Creates a Pascal object file that can be linked to from other Pascal procedures/functions
	-FRN / -NOFRN	Controls generation of floating point round instruction

Specify the Existence and Contents of the Source Listing

*	-L [object]	Controls existence of listing file
*	-XREF / noxref	Cross reference in source listing
	-EXPLIST / noexplist	Assembly code in source listing
	-OFFSET / nooffset	Offset map in source listing

Specify the Handling of Error and Statistics Information

-SILENT / nosilent	Suppresses Warning Messages
-STATISTICS / nostatistics	Prints compilation statistics
-STANDARD / nostandard	Checks for standard syntax

* Indicates options most useful to new users.

Prime-supplied defaults are underlined.

Specify the Source File

The source file is usually designated by pathname immediately after the PASCAL command. Alternatively, it may be given in an option.

> -SOURCE pathname or -INPUT pathname

Either of these can designate the source file to be compiled, instead of naming the file immediately after the PASCAL command. The following are equivalent:

PASCAL pathname

PASCAL -I pathname

PASCAL -S pathname

The pathname must not be designated more than once in the command line.

Specify the Existence and Properties of the Object Code

For a given source program, the compiler can produce a variety of object programs or none at all, depending on the options given. The areas open to programmer control are:

- Creation of the object file
- Storage allocation and addressing
- Compiler augmentation of the object code

Creation of the Object File: The -B option controls the existence and naming of the object file, but not the properties it will have.

> * -BINARY [object]

The object may be:

pathname	Object code will be written to the file pathname.
YES	Object code will be written to the file named B_program, where program is the name of the source file.

NO — No binary file will be created. Specified when only a syntax check or source listing is desired.

When no -B option is given, or -B without an object is given, -B YES will be presumed.

Storage Allocation and Addressing: By giving appropriate options, the user can cause compiled subprograms (procedures or functions) to accept ARRAY or RECORD formal parameters longer than a segment, and can determine the addressing mode (64V or 32I) to be used in the object file.

> -BIG / nobig

Determines the type of code generated for references to ARRAY or RECORD formal parameters in a subprogram.

-BIG: An ARRAY or RECORD formal parameter can become associated with any ARRAY or RECORD respectively.

nobig: An ARRAY or RECORD formal parameter can be associated only with an ARRAY or RECORD that does not cross a segment boundary.

See ARRAY or RECORD Type Formal Parameters in Section 9 for details.

> -64V / -32I

These determine the addressing mode to be used in the object code. 64V is a segmented virtual addressing mode for 16-bit machines. 32I is a segmented virtual mode which takes maximum advantage of the 32-bit architecture of Prime's more advanced models (P450 and up).

Augmented Object Code: When no augmented-code options are given, the source program is compiled statement by statement, and the resulting object code becomes the object file. Alternatively, the compiler can optimize the object code, and can add additional code to provide range checking, external procedure definition, floating point round instruction, or the capacity to run under the symbolic debugger.

> * -DEBUG / nodebug

Controls generation of code for the debugger.

-DEBUG: The object file is modified so that it will run under the symbolic debugger. Execution time is increased. The code generated will not be as highly optimized.

nodebug: No debugger code is generated.

Note

The DEBUG option is currently not supported by the Prime Pascal compiler.

> * -OPTIMIZE / -NOOPTIMIZE

Controls the optimization phase of the compiler.

-OPTIMIZE: The object code will be optimized. Optimized code runs more efficiently than non-optimized code, but takes somewhat longer to compile.

-NOOPTIMIZE: Optimization does not occur.

> -PRODUCTION / noproduction

Alternative option controlling code for the debugger.

-PRODUCTION: Similar to DEBUG, except that the code generated will not permit insertion of statement break points. Execution time is not affected.

noproduction: Production-type code is not generated.

Note

The PRODUCTION option is currently not supported by the Prime Pascal compiler.

> * -RANGE / -NORANGE

Controls error checking for out-of-bounds values of array subscripts and character substring indexes.

-RANGE: Error-checking code is inserted into the object file. Should an array subscript or character substring index take on a value outside the range specified when the referenced data item was declared, an error will be generated. Range checking decreases the efficiency of the generated code.

-NORANGE: Out-of-bounds values will not be detected. The program will be more vulnerable to errors, but will execute more quickly.

> -EXTERNAL / noexternal

Generates an external procedure definition.

-EXTERNAL: Similar to E+ compiler switch, except EXTERNAL option cannot be suppressed or resumed during compilation. (E+ switch is discussed at the end of this section.)

noexternal: No external procedure definition is generated.

> -FRN / -NOFRN

Controls generation of floating point round instruction.

-FRN: Causes an FRN instruction to be generated before every FST instruction in the code produced by the Pascal compiler. The FRN option improves the accuracy of single precision floating point calculations at some slight run-time performance expense.

-NOFRN: No FRN instruction will be generated.

Specify the Existence and Contents of the Source Listing

The Pascal compiler's primary output to the user is the source listing. When the -L option is given, a basic source listing is created, containing:

- Date and time of compilation
- Options in effect
- Source text
- External entry points
- Symbol-Table Listing
- List of errors

Additional options can be given, to cause additional data to be inserted into the source listing: a cross reference, offset map, or pseudo-assembly code listing may be included. If such an option is given but no source listing is specified, -L YES will be assumed.

> * -LISTING [object]

Controls creation of the source listing file. The object may be:

pathname	Listing will be written to the file pathname.
YES	Listing will be written to a file named L_program, where program is the name of the source file.
TTY	The listing will be printed at the user terminal.
SPOOL	The listing will be spooled directly to the line printer. Default SPOOL objects are in effect.
NO	No listing file will be generated.

When no -L option is given, -L NO will be presumed. When -L is given with no object, -L YES will be presumed.

> * -XREF / noxref (Implies -L)

Controls generation of a cross reference.

-XREF: A cross reference will be appended to the source listing. A cross reference lists, for every variable, the number of every line on which the variable was referenced.

noxref: No cross reference will be generated.

> -EXPLIST / noexplist (Implies -L)

Inserts a pseudo-assembly code listing into the source listing.

-EXPLIST: Each statement in the source will be followed by the pseudo-PMA (Prime Macro Assembler) statements into which it was compiled. For information on PMA, see The Assembly Language Programmer's Guide.

noexplist: No assembler statements are printed.

> -OFFSET / nooffset (Implies -L)

Appends an offset map to the source listing.

-OFFSET: An offset map is appended to the source listing. For each statement in the source program, the offset map gives the offset in the object file of the first machine instruction generated for that statement.

nooffset: No offset map is created.

Specify the Handling of Error and Statistics Information

Level 1 error messages (warnings) can be suppressed if desired. Compiler statistics can be printed at the terminal after each phase of compilation, but not to a user file other than a COMOUTPUT file.

> -SILENT / nosilent

Suppresses WARNING messages.

-SILENT: Level 1 Error Messages will not be printed at the terminal, and will be omitted from any listing file.

nosilent: Level 1 Error Messages are retained.

> -STATISTICS / nostatistics

Controls printout of compiler statistics.

-STATISTICS: A list of compilation statistics is printed at the terminal after each phase of compilation. For each phase the list contains:

- o DISK: Number of reads and writes during the phase, excluding those needed to obtain the source file.
- o SECONDS: Elapsed real time.
- o SPACE: Internal buffer space used for symbol table, in 16K byte units.
- o PAGING: Disk I/O time.
- o CPU: CPU time in seconds, followed by the clock time when the phase was completed.

nostatistics: Statistics are not printed.

> -STANDARD / nostandard

Controls flagging of non-standard Pascal syntax.

-STANDARD: A warning will be generated for any syntax departures from the proposed ANSI Pascal.

nostandard: No such warning is generated.

OPTION ABBREVIATIONS

The PASCAL compiler options may be abbreviated, as follows:

The abbreviations -L, -B, -I, and -S stand for -LIST, -BINARY, -INPUT, and -SOURCE, respectively, regardless of what other abbreviations are used.

Except where the above rule takes precedence, the abbreviation for any compiler option is the shortest string of leftmost characters from the option's name that uniquely identify the option. Any number of additional characters, up to the complete name, may also be given.

These rules produce the abbreviations shown in Table 2-2. The table is also intended to provide a quick alphabetical reference for those already familiar with the compiler options.

Table 2-2. Summary of Compiler Options and Abbreviations. (Defaults are underlined.)

Option	Abbreviation	Significance
-BIG	-BIG	Boundary-spanning code
-BINARY	-B	Create object file
-DEBUG	-DE	Debugger code
-EXPLIST	-EX	Expanded source listing
-EXTERNAL	-EXT	Generate external procedure definition
-FRN	-FRN	Generate floating point round instruction
-INPUT	-I	Designate source file
-LIST	-L	Create source listing
-NOFRN	-NOFRN	Don't generate FRN instruction
-NOOPTIMIZE	-NOOP	Don't optimize object code
-NORANGE	-NOR	Don't check subscript ranges
-OFFSET	-OF	Offsets in source listing
-OPTIMIZE	-OP	Optimize object code
-PRODUCTION	-P	Generate production code
-RANGE	-R	Check subscript ranges
-SILENT	-SI	Suppress warning messages
-SOURCE	-S	Designate source file
-STANDARD	-STAN	Flag non-standard Pascal syntax
-STATISTICS	-ST	Print compiler statistics
-XREF	-X	Generate cross-reference
-32I	-3	Produce 32I mode code
-64V	-6	Produce 64V mode code

COMPILER SWITCHES

The compiler functions can also be controlled through the use of compiler switches specified within the source program.

A compiler switch is written as a comment -- text enclosed in delimiters (* *) or {} -- with a dollar sign as the first character. Immediately following the $, a letter designates the specific switch and a + or - sign thereafter indicates the turning on or off of the switch. This format up to the +/- sign must be followed strictly or the switch will be ignored by the compiler. Any note, if desired, may be written between the +/- sign and the *) or }. Examples:

{$L+}

{$A - Compiler will ignore this switch.}

{$A- Compiler recognizes this switch.}

(*$E+, $L- Compiler will only recognize the first switch.*)

(*$E+*) (*$L- Compiler will recognize both E+ and L- switches.*)

Multiple switches, written as separate comments, can be used to control the compilation of a specific part of a program.

The available compiler switches, and their meanings are as follows:

Switch	Meaning	Default
A	Controls the generation of code used to perform array bounds checking at run-time. A- suppresses the generation; A+ resumes it.	A+
L	Controls the printing of source lines to the listing file at compile time. L- suppresses the printing of source lines (source text); L+ resumes it.	L+
E	Controls the definition of globally defined procedures and variables. Pascal procedures/functions can be separately compiled by including {$E+} at the beginning of the module. (A detailed discussion is presented at the end of Section 9.)	E-

SECTION 3

LOADING AND EXECUTING PROGRAMS

The PRIMOS SEG utility loads and executes all Pascal programs. This section summarizes normal loading and execution. The loading concept is described in more detail in The Prime User's Guide. For extended loading features, as well as a complete description of all SEG commands, including those for advanced system-level programming, refer to The LOAD And SEG Reference Guide.

LOADING PROGRAMS

Normal Loading

The following basic procedure loads most programs:

1. Invoke the SEG loader with the SEG command. (A "#" sign will be the prompt symbol.)

2. Enter the SEG-level LOAD command to start the load subprocessor and to set up the runfile (i.e., LO #filename). (A "$" sign will appear as the next prompt symbol.)

3. Use the load subprocessor's LOAD command to load the object files in the following order:

   - The object file of the main program (i.e., LO B_filename)
   - The object files of any separately compiled subroutines (preferably in order of A calls B calls C, etc.)

4. Use the load subprocessor's LIBRARY command to load subroutines called from libraries in the following order:

   - Pascal library (i.e., LI PASLIB)
   - Other Prime libraries, if required (i.e., LI filename), such as sort library VSRTLI, applications library VAPPLB, etc.
   - Standard (FORTRAN) library (i.e., LI)

5. At this point, the user should receive a LOAD COMPLETE message. If the message is absent, use the MAP 4 command to identify the unsatisfied references and load them. If the unsatisfied references are caused by misspelled subroutine names, the user should initialize and repeat the load. In the unlikely event some other SEG error message appears, refer to The LOAD And SEG Reference Guide for the probable cause and correction.

6. The QUIT command saves the runfile and exits from the utility.

Example:

```
OK, SEG
[SEG rev 17.5]
# LO #MAIN
$ LO P_MAIN          main program first
$ LO R_SUBR          separately compiled subroutine next
$ LI PASLIB          Pascal library
$ LI                 standard (FORTRAN) library
LOAD COMPLETE        loader indicates all references are satisfied
$ G                  save the runfile and return to PRIMOS command
                     level
OK,
```

EXECUTING LOADED PROGRAMS

Execution of Runfiles

Execution is performed at the PRIMOS level using the SEG command:

SEG #filename

where #filename is the filename (or pathname) of a SEG runfile. SEG loads the runfile into segmented memory and begins execution of the program.

A shortcut to saving and executing a loaded program is available. Immediately after receiving the LOAD COMPLETE message, enter the load subprocessor's XECUTE command. This command will then save the loaded program and start executing the program.

Upon completion of program execution, control returns to PRIMOS command level.

Run-time Error Messages

An alphabetic list of the Pascal run-time error messages are provided in Appendix A of this document. System run-time error messages are in The Prime User's Guide.

SECTION 4

PASCAL LANGUAGE ELEMENTS

DEFINITIONS

The terms defined below are to be used repeatedly throughout the book. Some of these terms will be further described in the sections noted. Many other terms are defined in later sections.

Term	Definition
Program	A main program consists of a heading and a block, and ends with a period. (See Section 5.)
Program Unit	A program unit can be a main program, a procedure, or a function.
Subprogram	A subprogram is either a procedure or a function. It consists of a heading and a block, and ends with a semicolon ";". (See Section 9.)
Heading	A heading gives a program unit a name and lists its parameters. (See Sections 5 and 9.)
Object	An object is an identifier or a label used in a program unit. (Identifiers and labels are defined later in this section.) Objects can be called data objects, or simply called data.
Block	A block is the body of a program unit. It consists of a sequence of declarations describing data objects to be used in the program unit, and a sequence of statements describing actions to be performed on these objects. (See Sections 5 and 9.) A program unit can have up to 64 levels of nesting of blocks within blocks. If block B is defined within block A, then B is called the "inner block" or "inner level", and A is called the "outer block" or "outer level". If block C is defined within block B, then B becomes an outer block to inner block C, but is still an inner block to block A. The outermost block (level) of a program is the program block (level) itself.

Global — The data objects declared in the outer block of a program unit are accessible at all inner levels of the program unit and are termed "global". If block B is defined in block A and block C is defined in block B, then an object declared in A is said to be global to B and C and an object declared in B is said to be global to C.

Objects declared at the program level (the outermost level) are global to all inner levels and have significance throughout the entire program.

Local — An object declared in a block, which is available or significant only within that block, is said to be "local" to that block. However, if block B is nested in block A, then objects local to A are global to B and have significance in both A and B.

Scope — The block in which an object is declared defines the "scope" of that object. In other words, the scope of an identifier or label is the portion of a program text in which the declaration or definition of the identifier or label is valid.

Actual Parameter — An actual parameter is a variable passed to a subprogram. Actual parameters appear in the parameter list of a procedure statement or a function designator. (See Section 9.)

Formal Parameter — A formal parameter is a variable appearing in the parameter list of a subprogram heading. When the subprogram is invoked, each formal parameter is associated with the actual parameter whose name appears in the corresponding position in the procedure statement or the function designator. A formal parameter can also be called dummy parameter. (See Section 9.)

PASCAL CHARACTER SET

The Prime Pascal character set consists of:

- 26 uppercase and 26 lowercase letters of the English alphabet (A to Z, a to z).
- 10 digits (0 - 9).
- 21 punctuation symbols. These symbols are used in single and in certain combinations to represent operators and delimit textual elements as described in Table 4-1.

Table 4-1. Pascal Punctuation Symbols.

SYMBOL	DESCRIPTION
+	Addition Identity Set Union
-	Subtraction Sign-inversion Set Difference
*	Multiplication Set Intersection
/	Division (real)
=	Equal to Set Equality Type-identifier and Type Separator
<	Less than
>	Greater than
[]	Subscript list or Set Constants Delimiters
.	Decimal Point Record Selector Program Terminator
,	Parameter or Identifier Separator
:	Variable Name and Type Separator Label and Statement Separator
;	Statement Separator Record Field Separator Declaration Separator
^	File or Pointer Variable Indicator
()	Parameter list, Identifier list, or Expression Delimiters
<>	Not equal to Set inequality

Table 4-1. Pascal Punctuation Symbols (cont.)

SYMBOL	DESCRIPTION
<=	Less than or equal to Set inclusion ("is contained in")
>=	Greater than or equal to Set inclusion ("contains")
:=	Assignment Operator
..	Subrange Specifier
{}	Comment delimiters
(* *)	Comment delimiters
'	Character-string delimiter (apostrophe)
&	Bitwise Integer AND (Prime Extension)
!	Bitwise Integer OR (Prime Extension)

KEYWORDS

Keywords are special symbols with fixed meanings and purposes, which the user cannot redefine. They can be used only as specified in the syntax for a Pascal program unit. Keywords may be written in lowercase letters, uppercase letters, or any combination of them. Lowercase letters will be interpreted the same as their uppercase counterpart. Table 4-2 lists all the available keywords.

IDENTIFIERS

Identifiers are names used in Pascal source program units to denote programs, constants, types, variables, procedures, or functions. Identifiers may be written in either lowercase or uppercase letters or any combination of them. The compiler will convert all lowercase letters to their uppercase counterpart for the purpose of identifier recognition.

A Pascal identifier can be a user-defined identifier or a standard identifier.

User-Defined Identifiers

User-defined identifiers are names supplied by the user. These names cannot be keywords.

A user-defined identifier must begin with a letter or a dollar sign "$", which may be followed by any combination of letters, digits, underscores "_", and other dollar signs. It may contain up to 32 significant characters in its spelling. An identifier greater than 32 characters in length will result in a severity 3 error at compile time.

Note

SEG limits the length of external names to 8 characters. Therefore, all external variable, procedure and function names should be less than or equal to 8 characters in length.

Table 4-2. Pascal keywords.

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

* Prime keyword

Standard Identifiers

Standard identifiers are names with predefined meanings and purposes. Any standard identifier may, if necessary, be redefined globally or locally by the user for another purpose. However, if an identifier is redefined, it cannot be used for its original purpose within the scope of that redefinition. For example, a user may create a variable named ABS. Then, however, the user would no longer be able to use the standard absolute value function ABS in the block containing the declaration of that variable. Table 4-3 lists all the available standard identifiers. Detailed descriptions are contained in appropriate sections of this book.

NUMERIC CONSTANTS

Pascal has two forms of numeric constants -- integer and real.

An integer is a whole number with an optional sign; it is an INTEGER type constant.

A real number has a fractional part; it is a REAL type constant. There are two ways of expressing real numbers.

1. In decimal notation, the number is expressed by an optional sign, a whole number part, a decimal point, and a fractional part. There must be at least one digit on each side of the decimal point.

2. In scientific notation, the number is represented by a value, followed by a letter E followed by an exponent. The value consists of an optional sign, one or more digits, and an optional decimal point and fractional part. The exponent must be an integer with an optional sign. The letter E is read as "times 10 to the power of". This is a convenient way to represent very large or very small numbers.

No comma may appear in a number. Examples:

Valid Integer	Invalid Integer
23	+40000 (Only 16-bit integers allowed)
+100	-32,768 (No comma allowed)

Valid Real Number	Invalid Real Number
-0.1	.1 (No digit to the left of the decimal point)
1e6 (1000000)	1. (No digit to the right of the decimal point)
5E-8 (0.00000005)	-3.0E-5.3 (Only whole number exponents allowed)
-87.35E+15 (-87350000000000000)	1,234E+20 (No comma allowed)
-7.0e-6 (0.000007)	

Table 4-3. Standard Identifiers.

Constants

FALSE	TRUE	MAXINT

Types

INTEGER	BOOLEAN	REAL	CHAR	TEXT

Files

INPUT	OUTPUT

Directives

FORWARD	EXTERN*

Functions

ABS	EXP	SIN
ARCTAN	LN	SQR
CHR	ODD	SQRT
COS	ORD	SUCC
EOF	PRED	TRUNC
EOLN	ROUND	

Procedures

CLOSE*	PAGE	RESET
DISPOSE	PUT	REWRITE
GET	READ	WRITE
NEW	READLN	WRITELN

* Prime extension

LABELS

Labels are unsigned integers from 0 to 9999 inclusive. They are used to mark the intended destination of a GOTO statement. The destination statement is prefixed by a label followed by a colon. For example, 120: WRITELN (' END').

CHARACTER-STRINGS

A character-string is a character or a sequence of characters enclosed by apostrophes. Character-strings consisting of a single character are CHAR type constants. Character-strings consisting of more than one character are string type constants (i.e., "ARRAY [1..n] OF CHAR", where "n" is the number of characters represented by the string). (Data types are described in Section 6). To include an apostrophe character in a string, double the apostrophe.

Examples:

```
''''
'A'
';'
'THIS IS A STRING'
'Pascal'
'Don''t give up the ship.'
```

DECLARATIONS AND STATEMENTS

Declarations describe data objects to be executed in a program unit; statements perform explicit actions on the declared objects. Declarations must precede statements in the program text. (See Sections 5 and 8 for detailed discussions.)

LINE FORMAT

The Pascal compiler ignores the formatting of source lines. A declaration or statement may start anywhere on a line. More than one declaration or statement may be written on a single line. However, a keyword, an identifier, or a number cannot be divided between lines.

COMMENTS, BLANKS AND ENDS OF LINES

Comments, blanks (except in character-strings), and ends of lines are considered to be separators. Separators must not appear in identifiers, keywords, or unsigned numbers. At least one separator must be placed between identifiers, keywords, or unsigned numbers which are not separated by one or more of the punctuation symbols given in

Table 4-1. One or more separators may occur anywhere in the program text except as noted.

A comment has the form:

{sequence of characters}

in which the characters may be any character except the right brace "}" or the character sequence "*)". In Pascal, comments may be placed anywhere blanks are allowed. Comments are inserted as notes which indicate the purpose of a program or a section of code; also, certain comments are used to enable or disable compiler switches (see Section 2).

On many terminals, the brace characters are not available, so Prime Pascal also allows a comment to be delimited by the character pairs (* *). Delimiters {} and (* *) can be interchanged. Starting and ending comment delimiters need not have the same form.

SECTION 5

PASCAL PROGRAM STRUCTURE

A standard Pascal program consists of a heading and a block, and ends with a period. The block may contain up to six different kinds of declarations and a sequence of statements enclosed in the keywords BEGIN and END. Figure 5-1 illustrates the general structure of a program.

PROGRAM HEADING

The main difference between standard Pascal and Prime Pascal in program structure is the heading. In Prime Pascal, a program heading is optional.

A heading has the general form:

PROGRAM identifier [([file-identifier-list])];

The keyword PROGRAM must be the first word of a program heading. It is followed by an identifier which is the name of the program and an optional file-identifier-list which is a list of files (separated by commas) used by the program. (Files are explained in Section 6.) Examples:

```
PROGRAM SAMPLE;   {The file-identifier-list may be omitted.}

PROGRAM Y(OUTFILE);  {It is not necessary to list all the files

PROGRAM X();         used by the program.}

PROGRAM findroot(INPUT, OUTPUT);
```

Note

The program heading, if present, is only checked syntactically by the Prime Pascal compiler. No check is made to see if the files named exist.

Figure 5-1. Program Diagram

BLOCK

A block is divided into two parts -- declaration and executable. The declaration part contains declarations which describe all data objects to be used in the program. The executable part, delimited by the keywords BEGIN and END, contains statements which specify the actions to be executed upon these declared objects. The general form of a block is:

```
 [LABEL declaration;]
 [CONST definition;]
 [TYPE definition;]
 [VAR declaration;]
 [FUNCTION declaration;] ...
 [PROCEDURE declaration;] ...
BEGIN
 [statement-1 [;statement-2]...]
END
```

Example 1:

```
PROGRAM EX1;
LABEL
   1;
CONST
   ONE = 1;
TYPE
   SMALL = 1..3;
VAR
   TINY:  SMALL;
PROCEDURE P(VAR X: SMALL);
   BEGIN
      X: = 1
   END; {Of procedure P}
BEGIN
   P(TINY);
   IF (TINY) <> ONE)
      THEN BEGIN
             WRITELN ('ERROR');
             GOTO 1
           END;
   IF (TINY = ONE)
      THEN WRITELN ('TINY = ', ONE)
1:
END.  {Of program EX1}
```

Example 2:

```
{This is the null program.}
 BEGIN
 END.
```

Declaration Part

The declaration part of a block is divided into six subparts -- label, constant, type, variable, procedure, and function parts. It must precede the executable part.

Label Declaration Part: The label declaration part specifies all labels that mark statements in the corresponding executable part. The label declaration part has the form:

```
LABEL label [,label]...;
```

The keyword LABEL heads this part.

Each declared label, which is an unsigned integer consisting of up to four digits, must be unique and mark only one statement in the executable part. However, if block B is nested in block A, a label declared in A is allowed to be redefined in B. Example:

```
PROGRAM TEST(OUTPUT);
LABEL 6;
      .
      .
      .
PROCEDURE REDEFINE;
  LABEL 6;
      .
      .
      .
  BEGIN ...
    GOTO 6; ...
  6: END; {of procedure REDEFINE}
BEGIN
      .
      .
      .
  REDEFINE; ...
  GOTO 6; ...
6: END. {of program TEST}
```

Constant Definition Part: All constants to be represented by names in a program must be defined in the constant definition part. The form of this part is:

```
CONST    identifier-1 = constant-1;
        [identifier-2 = constant-2;]...
```

The keyword CONST heads this part.

An identifier is the name to be associated with a specific constant. It will be used in place of the constant throughout the entire block containing the definition, unless the identifier is redefined in a contained block.

A constant is a fixed value which may be an integer or real number with optional sign, a character-string, or a constant identifier (possibly signed). A constant identifier is an identifier that has already been assigned a constant value.

Examples:

```
CONST
   BLANK = ' ';
   TAX_RATE = 0.05;
   MAX = 50;
   MIN = -MAX;  {MAX is a constant identifier.}
```

Type Definition Part: All constants and variables in a program have types. The type of a constant is determined by the syntax of that constant. The type of a variable, on the other hand, must be explicitly specified in the variable declaration part (explained later in this section).

Pascal provides five standard (predefined) data types -- INTEGER, REAL, CHAR, BOOLEAN, and TEXT. In addition, Pascal permits users to define new data types in the type definition part of a program. (Data types are discussed in detail in Section 6.)

The type definition part, which always begins with the keyword TYPE, has the form:

```
TYPE    type-identifier-1 = data-type-1;
      [type-identifier-2 = data-type-2;]...
```

A type-identifier is the name of a specific data-type; it will be associated with one or more variables in the variable declaration part.

A data-type is either a new data type recognized by Prime Pascal or a type-identifier that has already been associated with a new data type.

Examples:

```
TYPE
  LETTERS = 'A'..'Z' ;
  STRINGS = ARRAY [1..50] OF INTEGER;
  DaysOfWork = (MON,TUE,WED,THUR,FRI);
  STR = FILE OF CHAR;
  CH = LETTERS;     {CH and LETTERS denote the same type.}
```

Variable Declaration Part: A variable is a named data object that can assume different values during the execution of a program. Variables to be used in the program must be declared in the variable declaration part. The form of this part is:

```
VAR     identifier-1 [, identifier-2]... : data-type-1;
      [identifier-3 [, identifier-4]... : data-type-2;]...
```

The keyword VAR heads this part.

An identifier is the name of a variable contained in the program. The variable must be explicitly associated with a data-type which determines the range of values the variable can assume, the set of operations that can be performed on it, and the class of standard procedures and functions that can be used on it.

The data-type may either be one of the standard data types (INTEGER, REAL, CHAR, BOOLEAN, or TEXT) or be a type-identifier as defined in the preceding type definition part. Examples:

```
TYPE
  OPERATION_SIGNS = (PLUS, MINUS, TIMES);
  EXAM_SCORES     = 0..100;
  STRING          = ARRAY [1..15] OF CHAR;
  DATE_RECORD     = RECORD
                      MONTH: 1..12;
                      YEAR:  INTEGER
                    END;
  LETTER_SETS     = SET OF 'A'..'Z';
  INTEGER_FILE    = FILE OF INTEGER;
VAR
  OPERATORS: OPERATION_SIGNS;
  SCORES   : EXAM_SCORES;
  STRING1, STRING2: STRING;
  DATE     : DATE_RECORD;
  LETTERS  : LETTER_SETS;
  INTEGERS : INTEGER_FILE;

  ROOT1, ROOT2       : REAL;
  COUNTER            : INTEGER;
  FLAG               : BOOLEAN;
  FILLER             : CHAR;
  TEXTIN, TEXTOUT    : TEXT;
```

The type definition and variable declaration may be combined. Examples:

```
VAR
  OPERATORS : (PLUS, MINUS, TIMES);
  SCORES    : 0..100;
  STRING1, STRING2: ARRAY [1..15] OF CHAR;
  DATE      : RECORD
                MONTH : 1..12;
                YEAR  : INTEGER
              END;
  LETTERS   : SET OF 'A'..'Z';
  INTEGERS  : FILE OF INTEGER;
```

However, it is necessary to keep the type definition and variable declaration separate, if the variables are to be used as actual parameters. (See Section 7 for detailed discussion.)

The association of an identifier and its data-type is valid throughout

the entire block containing the declaration, unless the identifier is redefined in a contained block. Suppose that block B is contained in Block A. An identifier declared in A can be reassigned to a variable of any type local to B and this redefined association is valid throughout the scope of B.

Example 1:

```
PROGRAM  SAM1;
VAR V : INTEGER;
    .
    .
    .
PROCEDURE P1;
  VAR V : REAL;
      .
      .
      .
                              ;
```

Example 2:

```
PROGRAM SAM2;
TYPE T : (TIME, TAPE, TIRE);
VAR  V : T;
      .
      .
      .
PROCEDURE P2;
  TYPE T : ARRAY [1..5] OF INTEGER;
  VAR  V : T;
      .
      .
      .
```

Procedure and Function Declaration Parts: Procedures and functions are the two types of subprograms in Pascal. Every procedure or function must be declared in the procedure declaration part or the function declaration part respectively, before it can be used. If both a procedure and a function are used in a program, the function declaration must precede the procedure declaration. Procedures and functions are discussed in detail in Section 9.

Executable Part

The executable part, delimited by the keywords BEGIN and END, contains a sequence of statements which perform explicit actions on the data described in the declaration part of the block. The executable part has the form of a compound statement. All statements are discussed in detail in Section 8.

Example:

```
{Program Heading}
PROGRAM CONVERSION(INPUT, OUTPUT);

{Declaration Part}
VAR  CHARACTER : CHAR;
     NUMBER    : INTEGER;

{Executable Part}
BEGIN
  READ(CHARACTER);
  NUMBER := ORD(CHARACTER);
  WRITELN(' CHARACTER ', '''', CHARACTER, '''',
    ' EQUALS TO NUMBER ', NUMBER)
END.
```

%INCLUDE

The compiler directive %INCLUDE provides a means of directing the compiler to include the contents of a specified file into the program unit at compile time.

The general form of %INCLUDE is:

%INCLUDE '*filename*';

where *filename* is the name of the file to be incorporated into the program unit at the position of %INCLUDE. The filename can be a pathname if the included file does not reside in the current UFD.

An %INCLUDE directive can appear anywhere that a declaration or definition of the declaration part, or a statement of the executable part, can appear. An included file may contain additional %INCLUDEs, but commonly contains:

- Declarations that are common to more than one program unit
- Numeric key definitions, especially for the file management system and application library.

%INCLUDE directives can be nested up to seven levels.

The %INCLUDE directive is a Prime extension to standard Pascal. It is a Prime keyword.

Example:

```
PROGRAM  SAMPLE;
%INCLUDE 'VAR_FILE';     {Suppose that VAR_FILE contains
         .                a set of commonly used
```

```
                .                   varible declarations.)
                .
PROCEDURE  P1;
%INCLUDE 'VAR_FILE';
                .
                .
                .
```

SECTION 6

DATA TYPES

Every constant, variable, function, or expression in a program must have a data type. The data type determines the set of values a variable may assume, or a function or an expression may generate. The data type also determines which operations are performed on the values, and how these values are represented in storage.

The data type of a constant, variable, function, or expression is determined as follows:

- The data type of a constant is determined by the syntax of that constant.
- The data type of a variable or function result is explicitly declared in a variable or function declaration. See Section 5 for a discussion of the variable and function declarations.
- The data type of the result produced by an expression is determined by the operands and operators within that expression. See Section 7 for a detailed discussion.

Figure 6-1 summarizes the data types available in Prime Pascal. Each data type is described later in this section. The internal representations of data types are discussed in detail in Appendix B.

SCALAR DATA TYPES

Scalar data types are the basic data types in Pascal. All other data types must be built from the scalar data types.

Each scalar data type has a group of distinct values, called constants, which have a defined linear ordering. Thus, each scalar type is ordered. The total number of constants in a type T is called the cardinality of T. The cardinality provides a measure of the storage needed to represent a variable of the type T.

Scalar data types are divided into two classes: standard scalar data types and user-defined scalar data types. The standard scalar types are the predefined, built-in types provided by Pascal. The user-defined scalar types are the new data types created by the user; they must be defined in the program in order to be recognized by the Pascal compiler.

Figure 6-1. The Hierarchy of Data Types in Prime Pascal

The Standard Scalar Data Types

There are four standard scalar types -- INTEGER, REAL, BOOLEAN, and CHAR.

The INTEGER Type: The INTEGER type comprises a subset of the whole numbers (integers) which are the 16-bit, twos-complement, fixed-point binary numbers. The values of the INTEGER type are in the range of -32768 to +32767 or -2^{15} to $+2^{15}-1$. The ordinal number of an integer constant is the integer constant itself.

However, the user may also use 32-bit integers in a program. In this case, the user must declare these integers as the constants of a subrange of the INTEGER type, but not the INTEGER type itself. (The subrange type is discussed later in this section.) Example:

```
I = -33000..+66000  {This will define I as a 32-bit integer.}
```

Note

Unsigned comparison of integers is not supported as in OMSI Pascal.

There is a predefined Pascal constant called MAXINT whose value is the largest available integer constant of the INTEGER type on a specific computer. Thus, on a Prime computer, MAXINT is 32767. (-MAXINT is equivalent to -32767.)

Some examples of the INTEGER type constants are:

Valid

```
 32767
  +200
     0
MAXINT
   -11
```

Invalid

```
 32,767      {No comma allowed}
 40000       {Greater than the upper bound 32767}
-32769       {This number is a 32-bit integer.}
 32.00       {A valid real number but not an integer}
```

There are five arithmetic operators: +, -, *, DIV, and MOD (modulus), and six relational operators: =, <>, <, >, <=, and >= available for the INTEGER type. Table 4-1 in Section 4 gives a brief description of each operator. Section 7 gives a detailed discussion of all Prime Pascal operators.

There are four standard functions used frequently to produce integer results. I is any integer and R is any real number.

```
ABS(I)      {Absolute value}
SQR(I)      {Square}
TRUNC(R)    {R truncated}
ROUND(R)    {R rounded}
```

See Section 11 for more information on standard functions.

The REAL Type: The REAL type is a subset of the real numbers (decimal values). On Prime computers, the approximate range of real numbers is $-1*10^{38}$ to $+1*10^{38}$.

There are two methods of representing real constants -- the decimal notation and the scientific notation (see NUMERIC CONSTANTS in Section 4). The following are examples of the REAL type constants:

```
  Valid
    +12.0
  3.14159
-0.123456
     23E3
  -7.0E-5
+2.01E+20

  Invalid
       2.      {No digit to the right of the decimal point}
   .10101      {No digit to the left of the decimal point}
    3E8.5      {Only whole number exponent permitted}
```

There are four arithmetic operators: +, -, *, and /, and six relational operators: =, <>, <, >, <=, and >= applicable to the REAL type. For more information, see Section 7.

There are also standard functions that produce REAL type results. R is any real number and X is either an integer or real number.

```
ABS(R)      {Absolute value}
SQR(R)      {Square}
SIN(X)      {Sine}
COS(X)      {Cosine}
LN(X)       {Natural logarithm}
EXP(X)      {Exponential}
SQRT(X)     {Square root}
ARCTAN(X)   {Inverse tangent}
```

See Section 11 for more information on standard functions.

The BOOLEAN Type: The two values (true and false) of the BOOLEAN type are denoted by the standard constants TRUE and FALSE.

The ordinal numbers of the Boolean values, FALSE and TRUE, are the integer values 0 and 1 respectively. Thus, FALSE < TRUE is defined.

The six relational operators =, <>, <, >, <=, and >= operate on any of the four standard scalar types or the string type (discussed later in this section) to produce a Boolean result.

In addition, three Boolean operators (OR, AND, and NOT) can be applied only to Boolean values to produce a Boolean result. All operators are described in detail in Section 7.

Three Boolean functions (ODD, EOF, and EOLN) return Boolean values TRUE or FALSE. See Section 11 for more information.

Note

Pascal BOOLEAN type is not compatible with FORTRAN and FORTRAN 77 LOGICAL type, but is compatible with PL/I BIT type.

The CHAR Type: The CHAR type is a set of characters, including both printable (graphic) and non-printable (control characters). The standard character set used by Prime is the ANSI, ASCII 7-bit character set. Internally, each character in the Prime computer is represented by a unique octal value which establishes the collating sequence for the character set. Appendix C presents the ASCII character set and the associated octal values, which range from 200 to 377.

To indicate a constant of the CHAR type, place an apostrophe (a single quote) on each side of the character. To indicate an apostrophe, write it twice. Examples:

```
'A'
'7'
';'
''''
' '       {Blank is considered a "printable" character.}
```

Note

A constant of the CHAR type is always a single character. Constructs such as '123' or 'STRING' are not constants of this type but are constants of a more complex type called the string type which is described later in this section.

Each character constant externally corresponds to a unique integer value which is the ordinal number of the constant. The ordering relationship between any two character values is the same as between their ordinal numbers. The ordinal number is produced by the standard ordinal function ORD. Examples:

```
ORD('A') = 193              {Octal value 301}
ORD('a') = 225              {Octal value 341}
ORD('0') = 176              {Octal value 260}
ORD('1') - ORD('0') = 1     {Characters '0' through '9' can be
                             converted to integers 0 through 9 by
                             using this expression.}
```

The character function CHR is the inverse of ORD. Examples:

```
CHR(193) = 'A'
CHR(1 + ORD('0')) = '1'
```

There are two more standard functions particularly useful for processing character data -- PRED (predecessor function) and SUCC (successor function). When the argument of PRED and SUCC is of type CHAR, the functions can be defined as:

```
PRED(ch) = CHR(ORD(ch) - 1)
SUCC(ch) = CHR(ORD(ch) + 1)
```

Examples:

```
PRED('P') = 'O'
SUCC('P') = 'Q'
```

Functions are described in detail in Section 11.

The relational operators =, <>, <, >, <=, and >= are applicable on all character constants. For more information, see Section 7.

The User-Defined Scalar Data Types

There are two user-defined scalar types -- enumerated and subrange.

The Enumerated Types: An enumerated type defines an ordered set of values by enumerating the identifiers that denote these values.

To create an enumerated type, use the following type definition:

TYPE type-identifier = (identifier-1, identifier-2 [,identifier-3]...);

The identifier contained in the parentheses are the constants of the new enumerated type, and the type-identifier is the name of the new type.

The ordinal number is 0 for the first (leftmost) constant and is incremented by 1 for each successive constant. The largest allowable ordinal number of an enumerated type is 32767 on Prime computers. The ordering relationship between any two constants is the same as between their ordinal numbers. Examples:

```
TYPE COLOR = (RED, YELLOW, GREEN, BLUE, PINK);
     SEX = (MALE, FEMALE);
     FLAG = (FALSE, TRUE);
     DAYS = (MON, TUE, WED, THUR, FRI, SAT, SUN);
     MONTH = (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV,
              DEC);
```

These newly created data types are associated with variables by using the variable declarations:

```
VAR COLOURS: COLOR;
    S: SEX;
    F: FLAG;
    WEEKDAY: DAYS;
    MONTHS: MONTH;
```

The type definition and variable declaration may be combined. Example:

```
VAR S: (MALE, FEMALE);
```

If s1, s2 and s3 are valid statements, then the following examples, based on the above declarations, are meaningful statements:

```
WHILE WEEKDAY <= FRI DO s1;

FOR MONTHS := MAR TO SEP DO s2;

WEEKDAY := SUCC (WED);

F := TRUE;

IF COLOURS <> GREEN THEN s3;
```

The constants of enumerated types must not appear in more than one enumerated type. The following example is illegal:

```
TYPE FAMILY = (MOTHER, FATHER, SISTER, BROTHER);
     PARENTS = (FATHER, MOTHER);
```

The relational operators =, <>, <, >, <=, and >= are applicable on all enumerated types provided both operands are of the same enumerated types.

Three standard functions (SUCC, PRED, and ORD) apply to enumerated types. For example, given the following type definition:

```
TYPE SHAPE = (SQUARE, CIRCLE, RECTANGLE, TRIANGLE);
```

then

```
SUCC(CIRCLE) = RECTANGLE        {Successor of CIRCLE}

PRED(CIRCLE) = SQUARE           {Predecessor of CIRCLE}

ORD(CIRCLE) = 1                 {Ordinal number of CIRCLE}
```

The Subrange Types: A subrange type is a data type that comprises a specified range of any other already defined scalar data type, except type REAL.

To define a subrange type, use the following type definition:

```
TYPE type-identifier = lower-bound..upper-bound;
```

Both lower-bound and upper-bound are constants of the same standard scalar type (except REAL) or previously defined enumerated type, termed the base (host) type, and the lower-bound value must not be greater than the upper-bound value. The type-identifier is the name of the new data type that comprises only those base type constants within the lower- and upper-bound. Examples:

```
TYPE

 EXAMSCORE = 1..100;          {Subrange of INTEGER}
 DIGITS = '0'..'9';           {Subrange of CHAR}
 LETTERS = 'A'..'Z';

 DAYS = (MON, TUE, WED, THUR, FRI, SAT, SUN);  {Enumerated type}
 WEEKDAYS = MON..FRI;         {Subrange of DAYS}

 MONTHS = (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC);
 VACATION = JUN..SEP;         {Subrange of MONTHS}
 FIRST_TERM = JAN..MAY;       {Subrange of MONTHS}
```

Once the new data types are defined, they will be associated with appropriate variables by variable declarations. Example:

```
VAR
  SCORES : EXAMSCORE;     {EXAMSCORE is a subrange of INTEGER
                           defined in the first example above.}
```

The type definition and variable declaration may be combined. Example:

```
VAR SCORES : 0..100;
```

Based on the above example, the assignment SCORES := 95 is permissible, but SCORES := 101 is not. Any attempt to assign an out-of-range value to a subrange variable will cause a run-time error.

All the operations, functions, and procedures defined for the associated scalar type (base type) may be applied to the subrange type.

Constants of subranges of type INTEGER can either be 16-bit or 32-bit twos-complement, fixed-point binary numbers. Examples:

```
TYPE NUMBERS = -33000..+33000;      {32-bit binary numbers}
     LIMITS = 40000..60000;         {32-bit binary numbers}
     YEARS = 1700..1900;            {16-bit binary numbers}
```

STRUCTURED DATA TYPES

A structured data type is characterized by the type(s) of its components and by its structuring method. A component may have a scalar or structured data type. Although a structured data type can be quite sophisticated, it is ultimately built up from the scalar data type(s).

There are four basic structured data types -- ARRAY, RECORD, SET and FILE. Each of these new data types must be defined by the user in a type definition.

Caution

> The keyword PACKED used in type definitions does not have any effect. However, use of PACKED is not advised, and will generate a severity 1 error (warning) at compilation.

The ARRAY Types

An ARRAY type is a structure consisting of a fixed number of components (elements) which are all of the same type, called the base type. Elements of the entire structure are associated with a single name, called the array name. Array elements can be selected at random and are equally accessible. To denote an individual array element, follow the array name by the so-called index in square brackets [ ]. This index is to be a value of the type defined as the index type of the array. Therefore, the definition of an ARRAY type specifies both the base type and the index type. The general form is:

> TYPE type-identifier = ARRAY [t1] OF t2;

The type-identifier is the name of the new ARRAY type. t1 specifies the index type whose values are to be used in the index field. The index type must be a scalar data type other than REAL. t2 specifies the base type of the array elements. The base type can be any data type, even a structured type. If the base type has c values and the index type has values in the range of m (lower-bound) to n (upper-bound), then the cardinality of the ARRAY type is "c raised to the power (n-m+1)."

Example 1:

```
TYPE SAMPLE1 = ARRAY[1..100] OF REAL;
VAR R : SAMPLE1;
```

These declarations indicate that R will be a 100-element array of REAL. The first element will be accessed by R[1], the second by R[2],..., and the hundredth by R[100].

Example 2:

```
TYPE DAYS = (MON, TUE, WED, THUR, FRI, SAT, SUN);
     SAMPLE2 = ARRAY [DAYS] OF INTEGER;
VAR D : SAMPLE2;
```

These declarations indicate that D will be a 7-element array of INTEGER. The first element will be referenced by D[MON], the second by D[TUE],..., and the seventh by D[SUN].

Example 3:

```
TYPE EXAMSCORE = 1..100;
          .
          .
          .
VAR STUDENTSCORE : ARRAY[1..50] OF EXAMSCORE;
          .
          .
          .
BEGIN
          .
          .
          .
STUDENTSCORE[1] := 98;
          .
          .
          .
END.
```

The String Types: A line of text can be represented as an array of characters. This particular ARRAY type is called the string type. The definition of a string type on Prime computers is:

TYPE type-identifier = ARRAY [t1] OF t2;

where type-identifier is the name of the new string type; t1 is a subrange of type INTEGER with a lower bound of 1 and t2 is type CHAR. A string literal (a string of characters enclosed in two single quotes) is restricted in length within 2 to 256 characters.

Note

Although Prime Pascal does not support the keyword PACKED in type definitions, an ARRAY[1..n] OF CHAR is always stored as a packed ARRAY[1..n] OF CHAR on Prime computers.

Example 1:

```
TYPE STRING1 = ARRAY [1..10] OF CHAR;
VAR STRING2 : STRING1;
          .
          .
          .
BEGIN
          .
          .
          .
    STRING2 := 'ABCDEFGHIJ';
          .
          .
          .
    STRING2 := 'AB';              {This is an invalid assignment.
          .                        The string must contain 10
          .                        characters.}
          .
END.
```

Example 2:

```
          .
          .
          .
TYPE
   LENGTH = 1..30;
   STRING = ARRAY [LENGTH] OF CHAR;
VAR
   ALPHA : STRING;
   I : LENGTH;
          .
          .
          .
BEGIN
          .
          .
          .
   FOR I := 1 TO 30 DO READ (ALPHABET);
          .
          .
          .
END.
```

The Multidimensional Arrays: As defined in the previous section, the base type of an array can be any data type. If the base type of an array is an ARRAY type or a sequence of two or more ARRAY types, the array can be called a multidimensional array. Example:

```
VAR
  SAMPLE : ARRAY [BOOLEAN] OF ARRAY [1..10] OF ARRAY [SIZE] OF REAL;
                        {Assume that SIZE has already been defined.}
```

The above declaration can be simplified to more convenient forms:

```
VAR
  SAMPLE : ARRAY [BOOLEAN, 1..10, SIZE] OF REAL;

                              or

  SAMPLE : ARRAY [BOOLEAN] OF ARRAY [1..10, SIZE] OF REAL;

                              or

  SAMPLE : ARRAY [BOOLEAN, 1..10] OF ARRAY [SIZE] OF REAL;
```

In general, to create a multidimensional array, use the following type definition:

```
TYPE type-identifier = ARRAY [t1, t2,...] OF base-type;
```

where t1, t2, etc. are index types. If n index types are specified, the ARRAY type is called n dimensional and an array element is designated by n indices. Example:

```
CONST
   NUM_OF_CLASSES = 3; {3 classes}
   NUM_OF_STUDENTS = 20; {20 students in each class}
   NUM_OF_EXAMS = 4; {Each student took 4 exams}
TYPE
   SCORE = ARRAY [1..NUM_OF_CLASSES, 1..NUM_OF_STUDENTS,
                  1..NUM_OF_EXAMS] OF INTEGER;

VAR
   StudentScore : SCORE;
```

Using the example above, StudentScore[2, 10, 3] would designate the third exam of the tenth student in class number 2.

Similarly, StudentScore[3, 20, 4] would designate the fourth exam of the twentieth student in class number 3.

An array can have up to eight dimensions. The maximum size of an array, however, depends on the declaration of the array. If an array is declared using the EXTERN attribute, then the maximum size of the array may be up to 2^{31} words long. Otherwise, the maximum size of an array is 64K words long.

The RECORD Types

A RECORD type is a structure consisting of a fixed number of components (elements) which can have different data types. Elements of a RECORD type are called fields and each element has a name, called the field-identifier. To define a RECORD structure, use the following type definition:

```
TYPE record-identifer = RECORD
                          field-identifier-1: type;
                                    .
                                    .
                                    .
                          field-identifier-n: type
                        END;
```

where record-identifier is the name given to the entire structure. Each field-identifier and its associated type (which can be any type, even another RECORD type) is listed between the keywords RECORD and END. If the number of values in each of the fields is F1, F2,...,Fn, then the cardinality of the record type is (F1*F2*...*Fn).

Example 1:

```
TYPE PERSON = RECORD
                NAME : ARRAY [1..25] OF CHAR;
                AGE : 0..99;
                SEX : (MALE, FEMALE);
                SOC_NUM : INTEGER
              END;
```

Example 2:

```
TYPE
  CUSTOMER_RECORD =
    RECORD
      NAME : ARRAY [1..30] OF CHAR;
      ID_NUM : INTEGER;
      INVOICE_DATE : RECORD
                       MONTH : (JAN, FEB, MAR, APR,
                                MAY, JUN, JUL, AUG,
                                SEP, OCT, NOV, DEC);
                       DAY_OF_MON : 1..31;
                     END; {OF the INVOICE_DATE record}
      DISCOUNT, AMT_PAID : REAL
    END; {OF the CUSTOMER_RECORD}
```

Example 3:

```
TYPE DATE = RECORD
              DayOfWeek : (SUN, MON, TUE, WED,
                           THUR, FRI, SAT);
              Month     : (JAN, FEB, MAR, APR,
                           MAY, JUN, JUL, AUG,
                           SEP, OCT, NOV, DEC);
              DayOfMon  : 1..31;
              Year      : INTEGER
            END;

    FAMILY = (FATHER, MOTHER, BROTHER, SISTER);

 VAR DATE1, DATE2 : DATE;
     BIRTHDAY : ARRAY [FAMILY] OF DATE;
```

To access a particular record element, follow the name of the variable by a period and then by the name of the element:

```
record-variable.field-identifier
```

Using Example 3 above, if DATE1 is to contain the date:

Tuesday, July 15, 1980

the following assignment statements will be written:

```
DATE1.DayOfWeek := TUE;
DATE1.Month     := JUL;
DATE1.DayOfMon  := 15;
DATE1.Year      := 1980;
```

If the BIRTHDAY of SISTER is:

Saturday, December 6, 1975

The following assignment statements will be written:

```
BIRTHDAY[SISTER].DayOfWeek := SAT;
BIRTHDAY[SISTER].Month     := DEC;
BIRTHDAY[SISTER].DayOfMon  := 6;
BIRTHDAY[SISTER].Year      := 1975;
```

The references to elements in a record structure can be simplified by using the WITH statement. The general form of the WITH statement is:

WITH record-variable-1 [, record-variable-2]...DO statement

Within the statement after DO, record elements may be referred to by field-identifiers only. This form of record access allows the compiler to generate more efficient code and is suggested when a large number of components of a record are to be accessed.

Thus, the previous assignment statements can be rewritten as:

```
WITH DATE1 DO
   DayOfWeek := TUE;
   Month     := JUL;
   DayOfMon  := 15;
   Year      := 1980;

        and

WITH BIRTHDAY[SISTER] DO
   DayOfWeek := SAT;
   Month     := DEC;
   DayOfMon  := 6;
   Year      := 1975;
```

Note

The WITH statement is described in detail in Section 9.

Records with Variants: In Pascal, records of the same RECORD type need not have the same fields (components). In most cases, each of these records can be divided into two parts -- a part with fields common to all these records called the fixed part and a part with fields varying from record to record called the variant part. The fixed part must precede the variant part.

To define records with variants, use the following RECORD type definition:

```
TYPE  record-identifier =
        RECORD
           [field-identifier-1: type;]...                {fixed part}
           [CASE [tag-field:] tag-type-identifier OF     {variant part}
              variant-1 [; variant-2]...]
        END;
```

The tag-type-identifier denotes the data type of the tag-field. The data type, called the tag-type, may be any scalar type (except REAL). All the variants must be of a type compatible with the tag-type. Variants may be listed in any order in the variant part.

The tag-field is the name of a special record field, called the discriminating field or the type discriminator. The value of the tag-field determines which variant is active in determining the value of the variant part. If the tag-field is not contained in the variant part, then each value of the tag-type indicates which variant is assumed by the record variable.

If there are n values in the tag-type, and the variant associated with the value i has Ti values, then the cardinality of the variant part is (T1+T2+...+Tn).

A variant has the general form:

```
case-constant-1 [, case-constant-2]...:
  ([field-id-1: type [; field-id-2: type]...])
```

Each case-constant corresponds to a value of the tag-field or to a constant of the tag-type. The type of each variant field denoted by a field-id can be any data type. field-id cannot be duplicated, even in different variants. In Prime implementation, the largest variant field is allocated and all other variant fields are overlaid.

Note

The CASE clause is different from the CASE statement. The CASE statement is discussed in Section 5.

Example 1:

```
TYPE PERSON = RECORD
                L_NAME, F_NAME: ARRAY[1..20] OF CHAR;
                AGE: 0..100;
                SEX: (MALE, FEMALE);
                CASE MARRIED: BOOLEAN OF
                  TRUE: (SPOUSE_NAME: ARRAY[1..40] OF CHAR;
                         SPOUSE_AGE: 0..100);
                  FALSE: ( )
              END;
```

If a person is married, the field MARRIED, called the tag-field, will be TRUE, and two additional fields, called variant fields, will exist -- SPOUSE_NAME and SPOUSE_AGE. These variant fields will not exist if MARRIED is FALSE.

Example 2:

```
TYPE PERSON = RECORD
                L_NAME, F_NAME: ARRAY[1..20] OF CHAR;
                AGE: 0..100;
                SEX: (MALE, FEMALE);
                MARRIED: BOOLEAN;
                CASE BOOLEAN OF
                  TRUE: (SPOUSE_NAME: ARRAY[1..40] OF CHAR;
                         SPOUSE_AGE: 0..100);
              END;
```

Although this is a valid example of a RECORD type definition, its usage is not advised. The declaration of the tag-field in the CASE clause and the definition of every possible value of the tag-field as shown in example 1 give better program security and run-time diagnostics.

Example 3:

```
TYPE
  SHAPE = (POINT, LINE, CIRCLE);
  FIGURE = RECORD
             CASE TAG: SHAPE OF
               LINE: (M, B: REAL);
               CIRCLE: (A, C: REAL; RADIUS: REAL);
               POINT: (Xo, Yo: REAL)
           END;
VAR
  V : FIGURE; ...
BEGIN
    .
    .
    .
  V.TAG := LINE;
    .
    .
    .
  TAG := LINE;     {This assignment is invalid.}
    .
    .
    .
END.
```

Example 4:

```
TYPE
  Data = (Integer, Bool, Ch);
  DataType = RECORD
               CASE Data OF
                 Integer: (IntValue: INTEGER);
                    Bool: (BoolValue: BOOLEAN);
                      Ch: (ChValue: CHAR)
             END;
VAR
  DataValue: DataType; ...
BEGIN
  DataValue.IntValue := 100;
                .
                .
                .
  DataValue.BoolValue := TRUE;
                .
                .
                .
  DataValue.ChValue := 'A';
                .
                .
                .
END.
```

Example 5:

```
TYPE
  EMPTY = RECORD    {The record EMPTY contains no fields; therefore,
          END;       it has a null value.}
  .
```

The SET Types

A value of a SET type defines a set of elements chosen from a collection of elements of the same data type, termed the base type. The set is called the powerset of its base type. A base type can be any scalar data type other than REAL. To create a SET type, use the following type definition:

TYPE type-identifier = SET OF base-type;

The type-identifier is the name of a new SET type. All SET types are restricted to 256 elements on Prime computers. That is, the ordinals of elements in a set must lie in the range 0 to 255. Example:

```
TYPE LETTERS = SET OF 'A' .. 'Z';  {26 elements}
```

Variables of type LETTERS are declared in the variable declaration part:

```
VAR VOWELS, LIST, EMPTY, CH : LETTERS;
```

Each variable above is a set whose members are chosen from the alphabetic characters 'A' to 'Z'. Set members are set constants which are always presented in a pair of square brackets []. Values of set constants can be assigned to the variables by following assignment statements:

```
VOWELS := ['A', 'E', 'I', 'O', 'U'];     {Set numbers can be in
CH     := ['B', 'C', 'A'];                arbitrary order.}

EMPTY  := [ ];                           {A set may have no members
                                          at all, it is called the
                                          "empty set".}

LIST   := ['F'..'P'];                    {If the set members are
                                          consecutive values of the
                                          base type, only the first
                                          and last need be
                                          specified.}
```

There are three set operators that operate on sets to produce new sets.

+	Set union
-	Set difference
*	Set intersection

Examples:

['Q'] + ['P', 'Q'] is ['P', 'Q']

['A', 'B', 'E', 'F'] - ['B', 'C', 'D'] is ['A', 'E', 'F']

['E', 'I', 'O'] * ['A', 'E'] is ['E']

There are four relational operators that compare sets. The result of the comparison is a Boolean value.

=	Set equality
<>	Set inequality
<=	"Is included or contained in"
>=	"Contains"

Examples:

['A', 'B'] = ['B', 'C'] is false

['A', 'B'] <> ['B', 'C'] is true

['B'] <= ['B', 'C'] is true

['A'..'Z'] >= ['M'..'S'] is true

There is one more relational operator IN that tests set membership of a specified set. Examples:

'I' IN ['A', 'E', 'I', 'O', 'U'] is true

'I' IN ['P', 'S', 'X'] is false

Operators and operations are discussed in detail in Section 7.

The FILE Types

A FILE type is a structure consisting of a sequence of components (elements) which are all of the same type. The total number of elements, called the length of the file, is not fixed at the time the type is defined. This is a clear distinction between the file structure and the array structure. File elements cannot be selected at random and are not equally accessible. They must be accessed sequentially, one at a time. This is another distinction between the file and the array. A file with no elements is called an empty file. Pascal files are embodied as PRIMOS files.

To define a FILE type, use the following type definition:

```
TYPE type-identifier = FILE OF base-type;
```

where base-type specifies the data type of each individual element of the file; it must neither be a FILE type nor a structured type with a FILE component. The new file is called "type-identifier". Example:

```
TYPE
  IntegerFile = FILE OF INTEGER;    {A sequence of internal binary
                                     integers}
  ArrayFile = FILE OF ARRAY [1..15] OF REAL;
                                    {A sequence of groups of 15
                                     internal binary real numbers}
  CharFile = FILE OF CHAR;          {A sequence of characters -
                                     a textfile}
```

The declaration of a file variable f will automatically create a buffer variable, indicated by f^, of the FILE type. A buffer variable can be considered as a window through which one can read from or write into a file. Example:

```
VAR
  I: IntegerFile;     {The FILE type of each variable has already
  A: ArrayFile;        been defined in the previous example.}
  C: CharFile;
```

The variable declarations above will create three buffer variables I^, A^, and C^. Only the element currently in the buffer variable is immediately accessible.

Five I/O procedures -- RESET, GET, REWRITE, PUT, and CLOSE, and one Boolean function EOF (End-Of-File) control and test the files and the file variables. They are described in detail in Section 10.

The TEXT File: There is one especially important FILE type (FILE OF CHAR) that deserves special attention. A "FILE OF CHAR", frequently called a "textfile", is a file consisting of printable characters, including integers and real numbers. A "FILE OF INTEGER" is a file of internal binary integers and a "FILE OF REAL" is a file of internal binary real numbers; they are not readily accessible to the general user. Textfiles, on the other hand, can be created, updated, and inspected with Prime's text EDITOR. Therefore, the interface between computers and their users is almost always by textfiles.

A textfile is denoted by the standard identifier TEXT and is predefined as follows:

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

A textfile may be subdivided into variable length lines. Each line in the file is separated from the next by an ASCII control character -- CR (Carriage Return) or LF (Line Feed).

The I/O procedures READ, READLN, WRITE, WRITELN, and PAGE and the Boolean function EOLN (End-Of-Line) control and test textfiles only. They are described in detail in Section 10.

Example:

```
VAR
  TEXTIN, TEXTOUT : TEXT;
  CH : CHAR;
       .
       .
       .
BEGIN
  WHILE NOT EOLN(TEXTIN) DO
       .
       .
       .
    READ(TEXTIN, CH);
    WRITE(TEXTOUT, CH);
       .
       .
       .
    READLN(TEXTIN);
    WRITELN(TEXTOUT);
       .
       .
       .
END.
```

The Standard Textfiles: The standard identifiers INPUT and OUTPUT are file variables associated with an input textfile and an output textfile respectively. They are predefined as follows:

```
VAR INPUT, OUTPUT : TEXT;
```

In Prime's implementation, the standard textfiles INPUT and OUTPUT provide that the input data are accepted from and the output data are displayed on the user's terminal. All other files provide that the input data are obtained from and the output data are stored into disk files.

In the following list, the procedure or function on the left column is equivalent to the corresponding procedure or function on the right column.

Procedure	Equivalent Procedure
READ(ch)	READ(INPUT, ch)
READLN	READLN(INPUT)
WRITE(ch)	WRITE(OUTPUT, ch)
WRITELN	WRITELN(OUTPUT)
PAGE	PAGE(OUTPUT)

Function	Equivalent Function
EOF	EOF(INPUT)
EOLN	EOLN(INPUT)

Note

When the procedures and functions on the left column above are applied to textfiles other than the standard textfiles INPUT and OUTPUT, the file variable must be the first parameter.

The procedures RESET and REWRITE must not be applied to the textfiles INPUT and OUTPUT and the result of doing so is undefined. For more information, see Section 10.

THE POINTER TYPE

A scalar or structured type variable is declared in the variable declaration part of a block and is accessible by its identifier. All the necessary memory is allocated for the variable at the time the block is about to begin execution. The memory of the variable (or simply, the variable) exists during the entire execution of the block. Therefore, this variable is called an automatic (or automatically allocated) variable. However, variables declared at the program level are allocated in static storage (linkage area); these variables are called static variables.

A variable accessed by a pointer, on the other hand, is created and destroyed dynamically during the execution of the block. Accordingly, this variable is called a dynamic variable.

Dynamic variables are not explicitly declared in variable declarations and are not referenced directly by identifiers. Instead, they are referenced indirectly by pointers. A pointer is the storage address of a newly created dynamic variable which is created by the predefined procedure NEW.

Pascal provides dynamic variables through the following type definition:

```
TYPE type-identifier = ^base-type;
```

The type-identifier is the name of a POINTER (dynamic) data type whose pointers will point to elements of the specified base-type. However, there is a special pointer constant, termed NIL, which is always an element of a POINTER type and points to no element at all. The POINTER type "type-identifier" is said to be bound to the base-type. Example:

```
TYPE POINTER = ^INTEGER;
VAR P : POINTER;
```

P is a pointer variable or a reference variable bound to an element of the INTEGER type, or say, P points to a variable of the INTEGER type. P^ is the actual variable being pointed to.

There are four procedures, called dynamic allocation procedures, that create and destroy dynamic variables.

NEW(p)	Creates (or allocates) a new dynamic variable. A pointer to this new variable is assigned to the pointer variable P.
DISPOSE(p)	Indicates that the storage occupied by the variable P^ is no longer accessible. That storage becomes available for future use. p is then undefined. NEW(p) and DISPOSE(p) are complementary.
NEW(p, c1,...,cn)	Creates (or allocates) a new dynamic variable of record type with variants. A pointer to this new variable is assigned to the pointer variable p. The variants of the variable (tag-field) correspond to the case-constants c1,...,cn. The case-constants must be listed continuously and in the order of their declaration. They must not be changed during execution.
DISPOSE(p, c1,...,cn)	Indicates that the storage occupied by P^, which was allocated by the NEW procedure immediately above, is no longer accessible. That storage becomes available for future use. p is then undefined. The case-constants of both procedures must be identical.

Note

Prime supports approximately 14 segments (896K words) of dynamic storage.

Prime's implementation of DISPOSE performs a true form of garbage collection.

Example 1:

```
PROGRAM POINTER_SAMPLE(INPUT, OUTPUT);
VAR
  PTR : ^INTEGER;              {PTR is a pointer variable bound to
                                type INTEGER.}
  I : INTEGER;
BEGIN
  FOR I := 1 TO 10 DO
```

```
      BEGIN
        NEW (PTR);              {Allocates a variable of type INTEGER
                                 and stores its address in PTR.}
        PTR^ := I;              {PTR^ is the actual variable being
                                 pointed to.  A value of I is assigned
                                 to this new variable.}
            .
            .
            .
        DISPOSE (PTR)           {Destroys the variable PTR^ and
                                 returns its storage for future use.}
      END;
    WRITELN ('This is an example.')
  END.
```

Example 2:

```
  {This example intends to create a linked list (or a chain) to which
  elements can be added or deleted at random.}

  TYPE LINK = ^PERSON;
       PERSON = RECORD
                  NEXT : LINK;
                  NAME : CHAR
                END;
  VAR ROOT, P : LINK;
      I : INTEGER;
      CH : CHAR;
  BEGIN
        .
        .
        .
    ROOT := NIL;                {The link of the last person is NIL.}
    FOR I := 1 TO 50 DO
      BEGIN
        READ(CH);
        NEW(P);
        P^.NAME := CH;          {A name code is stored in NAME.}
        P^.NEXT := ROOT;        {The sequence of these two statements
        ROOT := P                is a general algorithm for inserting
                                 an element at the beginning of the
                                 list.}
      END;
            .
            .
            .
  END.
```

SECTION 7

EXPRESSIONS

An expression is a single operand or a combination of operands and operators which can be evaluated to produce a value.

OPERANDS

An operand may be any of the following:

- A variable
- An unsigned number
- A character string
- A constant identifier
- A function designator (explained in Section 9)
- NIL
- An expression
- A set

Examples:

```
15
(x+y+z)
SIN(x+y)
[RED, C, GREEN]
[1, 5, 10..19,23]
NOT P
I * J + 1
-N
```

OPERATORS

Operators modify an operand or combine two operands. Operators can be classified as arithretic, relational, set, Boolean, or integer.

Arithmetic Operators

An arithmetic operator specifies computation to be performed on its operands to obtain a single numeric value. The following are the binary and unary arithmetic operators and the data types of operands and results:

Operator	Operation	Type of Operands	Type of Result
Binary:			
+ - *	addition subtraction multiplication	integer or real	integer if both operands are integer; otherwise real
/	division	integer or real	real
DIV	division with truncation	integer	integer
MOD	modulo	integer	integer
Unary:			
+ -	identity sign-inversion	integer or real	same as operand

Relational Operators

The relational operators are used to compare values of data types -- scalar, string, pointer, or set. In any given comparison, both operands must be of the same type. The result of the comparison is a boolean value, true or false. The following are the legal relational operators and data types of operands:

Operator	Operation	Type of Operands
=	equality	set, scalar, pointer, or string
<>	inequality	
<	less than	scalar or string
>	greater than	
<=	less or equal set inclusion ("is contained in")	scalar or string set

>=	greater or equal set inclusion ("contains")	scalar or string set
IN	set membership	first (left) operand is any scalar type (except REAL), second (right) operand is a set of that type

Examples:

first, let

```
x := ['A', 'D', 'C', 'E']
y := ['A', 'E']
z := ['B']
```

then

```
x = ['A', 'B', 'C', 'D']    {true }
y <= x                      {false}
y <> x                      {true }
z IN x                      {true }
```

Set Operators

Set operators, listed below, operate on sets to produce new sets.

Operator	Operation	Type of Operands	Type of Result
+	set union	any SET type T	T
-	set difference		
*	set intersection		

Examples:

first, let

```
x := ['A', 'E', 'O']
y := ['I', 'U', 'O']
z := ['A', 'B']
```

then

```
w := x + y     {w is ['A', 'E', 'I', 'O', 'U']}

w := x - y     {w is ['A', 'E']}

w := x * z     {w is ['A']}
```

Note

Five relational operators, =, <>, <=, >=, and IN, also apply to the set data type; they produce Boolean results. See Relational Operators.

Boolean Operators

Boolean operators operate on Boolean values to produce a Boolean result, TRUE or FALSE. The operators are OR, AND, and NOT. In the following examples, P and Q are of type BOOLEAN.

OR: P OR Q is the logical inclusive ORing of P and Q.

P	Q	P_OR_Q
F	F	F
F	T	T
T	F	T
T	T	T

AND: P AND Q is the logical ANDing of P and Q.

P	Q	P_AND_Q
F	F	F
F	T	F
T	F	F
T	T	T

NOT: NOT Q negates the value of Q.

Q	NOT_Q
F	T
T	F

Integer Operators

Integer operators & and ! are used to perform integer AND and integer OR functions, respectively. These functions take only integers as inputs and yield an integer value which is the bitwise ANDing or ORing of their inputs. These integer operators are a Prime extension to Pascal.

Operator Precedence

The precedence among operators determines the order in which expressions are to be evaluated. The precedence of operator is as follows:

1. NOT
2. *, /, DIV, MOD, AND, &
3. +, -, OR, !
4. =, <>, <, >, <=, >=, IN

ORDER OF EVALUATION

When operators having the same precedence appear successively in an expression, they can be evaluated in any order chosen by the compiler as long as the actual order of evaluation is algebraically equivalent to a left to right order of evaluation.

Parentheses may be used to override the normal evaluation order. An expression enclosed in parentheses is treated as a single operand, and evaluated first. When expressions are contained within a nest of parentheses, evaluation proceeds from the least inclusive set to the most inclusive set (i.e., inside out). Evaluation of expressions within parentheses takes place according to the normal order of precedence.

Examples:

```
7 + A * 2 - 5 DIV 3 + 4          (Numbers below the operators
  3   1   4     2     5           indicate the order in which
                                   the operations are performed.)

( 7 + A ) * 2 - 5 DIV ( 3 + A )
    1       2       4       3

( ( 7 + A ) * 2 - 5 ) DIV 3 + A
      1       2     3       4     5
```

SECTION 9

STATEMENTS

Statements specify algorithmic actions; they comprise the executable part of a program unit.

SUMMARY OF STATEMENTS

The various types of Pascal statements are listed below.

- Assignment Statement
- Procedure Statement
- Compound Statement
- Empty Statement
- Control Statements:

  REPEAT
  WHILE
  FOR
  IF
  CASE
  GOTO

- WITH Statement

ASSIGNMENT STATEMENT

An assignment statement assigns a value to a variable or a function identifier. The form of the statement is:

variable | function-identifier := expression

The assignment operator ":=" can be read "becomes". The expression on the right-hand side of the operator is evaluated and the value obtained becomes the current value of the variable or the function-identifier on the left-hand side of the operator.

A function-identifier is a function name. It may appear on the left-hand side of the assignment operator only within the function block of the function. (See Section 9.)

A variable is represented by its name. Values must have been previously assigned to all variables appearing in the expression on the right-hand side of the operator. Variables on the left-hand side may or may not have been assigned values previously. Example:

(If the user has made the following declarations in a program

```
VAR
  CH : CHAR;
  R  : REAL;
  NUMBER, F, I, J, K : INTEGER;
```

then the following assignment statements are valid.)

```
CH := '5';
NUMBER := ORD(CH) - ORD('0');
R := 123.3;
F := TRUNC(R) MOD 5;
I := F;
J := I + NUMBER;
K := J DIV 2;
I := SQR(K) - (1*J);
```

Assignment Compatibility

The data type of the expression on the right-hand side of the assignment operator is considered assignment-compatible with the data type of the variable or function-identifier on the left-hand side only if any of the six statements below is true. The expression is of type T2 and the variable or function-identifier is of type T1 (i.e., T1 := T2;).

- T1 and T2 are the same type, providing that neither T1 nor T2 is a FILE type nor a structured type with a FILE component.
- T1 is the REAL type and T2 is the INTEGER type.
- T1 and T2 are compatible scalar types (except REAL) and the value of the expression is in the closed interval specified by the type T1.
- T1 and T2 are compatible set types and all the members of the value of the expression are in the closed interval specified by the basetype of T1.
- T1 and T2 are string types with the same number of components.
- T1 is a subrange of T2, or T2 is a subrange of T1, or both T1 and T2 are subranges of the same type.

PROCEDURE STATEMENT

A procedure statement specifies the activation of the procedure denoted by the procedure-identifier in the statement. If the procedure has a list of formal parameters defined in the procedure declaration, then the procedure statement must contain a list of the corresponding actual parameters along with the procedure-identifier. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters. The actual and formal parameters must agree in number, type, and order.

The form of a procedure statement is:

```
procedure-identifier [(actual-parameter-list)]
```

The procedure-identifier is the name of the procedure. The optional actual-parameter-list enclosed in parentheses may contain one actual parameter or a list of actual parameters separated by commas. (For more information, see Section 9.) Examples:

```
PRINTHEADING

TRANSPOSE(A,N,M)

BISECT(FCT, -1.0, +1.0, X)
```

COMPOUND STATEMENT

A compound statement is a sequence of statements executed in the order in which they are written. The general form of a compound statement is:

```
BEGIN
   statement-1 [; statement-2]...
END
```

The keywords BEGIN and END are delimiters which designate the start and the end of the statements making up a compound statement. statement-1, statement-2, etc. can be any of the available Pascal statements. A compound statement can appear anywhere a single statement is allowed.

Example 1:

```
BEGIN
  Z := X;
  X := Y;
  Y := Z
END
```

Example 2:

```
BEGIN
  COUNTER := 0;
  READ (CHARACTER);
  WHILE (CHARACTER <> BLANK) DO
    BEGIN
      COUNTER := COUNTER + 1;
      READ (CHARACTER)
    END;
  WRITELN (' THE NUMBER OF CHARACTERS = ', COUNTER)
END
```

EMPTY STATEMENT

An empty statement denotes no action and, as its name implies, consists of no letters, digits or punctuation symbols.

Example 1:

```
CASE DAYS OF
  SUN:  ;  {An empty statement is right here.}
  MON, WED, FRI:  s1;
  TUE, THUR:  s2;
  SAT:  s3
END
```

Example 2:

```
IF DAY = SUN THEN  ELSE s1

                {An empty statement is right here.}
```

Example 3:

```
BEGIN
   READ(CH);
   WRITE(CH);  {The semicolon separates the WRITE procedure and an
                empty statement preceding END.}
END
```

CONTROL STATEMENTS

Statements are normally executed in the order of their appearance in a program unit. However, it is often necessary to interrupt the normal processing of statements for a special purpose, such as the repeated processing of a sequence of statements or the execution of one group of statements as opposed to another. Control statements are used to alter the normal sequential execution of statements.

There are three types of control statements: repetitive statements, conditional statements, and unconditional statements.

Repetitive Statements

Repetitive statements specify that certain groups of statements are to be executed repeatedly. There are three types of repetitive statements -- REPEAT, WHILE, and FOR. FOR is used when the number of repetitions does not depend on the statements within the loop. Otherwise, the REPEAT or WHILE statement must be used.

REPEAT Statement: The form of the REPEAT statement is:

REPEAT statement-1 [; statement-2]...UNTIL Boolean-expression

The statement (or the sequence of statements) between the keywords REPEAT and UNTIL is executed repeatedly until the Boolean-expression becomes true. The statement (or statement sequence) will be executed at least once, because the Boolean-expression is evaluated at the end of the cycle.

Example 1:

```
SPACE := ' ';
REPEAT
  READ(CH);
  WRITELN(CH);
UNTIL CH = SPACE
```

Example 2:

```
REPEAT
   K := I MOD J;
   I := J;
   J := K
UNTIL J = 0
```

Note

In a REPEAT loop, the beginning and the end of the statements to be executed repeatedly are explicitly designated by the keywords REPEAT and UNTIL. Therefore, it is not necessary to use the keywords BEGIN and END to bracket the statement sequence. However, if the bracketing pair BEGIN and END is

used, it is not wrong, just redundant.

WHILE Statement: The form of the WHILE statement is:

WHILE Boolean-expression DO statement

The statement, which may be any statement (including a compound statement), is executed repeatedly while the Boolean-expression is true. The Boolean-expression is evaluated at the beginning of each cycle. If its value is false initially, the statement will not be executed at all.

Example 1:

```
WHILE A[I] <> X DO I := I + 1
```

Example 2:

```
WHILE I>0 DO
  BEGIN IF ODD(I) THEN Z := Z * X;
     I := I DIV 2;
     X := SQR(X)
  END
```

Example 3:

```
WHILE NOT EOF(f) DO
  BEGIN
     PROCESS(f^);
     GET(f)
  END
```

A loop controlled by WHILE may be converted into a loop controlled by REPEAT. For example, the WHILE statement:

```
WHILE B DO BODY
```

is equivalent to:

```
BEGIN
   IF B THEN
     REPEAT
       BODY
     UNTIL NOT(B)
END
```

FOR Statement: A FOR statement causes a statement to be executed a specified number of times while a progression of values is assigned to a variable called the control variable of the FOR statement.

The general form of a FOR statement is:

```
FOR control-variable := initial-value TO final-value
   DO statement
```

The alternative form (for decreasing initial value) is:

```
FOR control-variable := initial-value DOWNTO final-value
   DO statement
```

The control-variable is denoted by its identifier which is declared in a variable declaration in the immediately enclosing block.

The statement constitutes the body of the FOR loop. It may be any statement, including a compound statement.

The control-variable is of any scalar type (except REAL). The initial-value and the final-value must be of a type compatible with the control-variable's type. Upon completion of the FOR statement, the control-variable is undefined.

When the FOR-TO form is used, the control-variable is tested to determine whether it is less than or equal to the final-value. If it is, the statement is executed, the control-variable is incremented by 1, and the cycle is repeated. This is illustrated by the examples below:

The FOR statement:

```
FOR v := e1 TO e2 DO body
```

is equivalent to:

```
BEGIN
temp1 := e1;
temp2 := e2;
IF temp1 <= temp2 THEN
   BEGIN
   v := temp1;
   body;
   WHILE v <> temp2 DO
      BEGIN
      v := SUCC(v);
      body
      END
   END
END
```

In the FOR-DOWNTO form, the control-variable is tested to determine whether it is greater than or equal to the final value. If it is, the statement is executed, the control-variable is decremented by 1, and the cycle is repeated. That is, the FOR statement:

```
FOR v := e1 DOWNTO e2 DO body
```

is equivalent to:

```
BEGIN
temp1 := e1;
temp2 := e2;
IF temp1 >= temp2 THEN
   BEGIN
   v := temp1;
   body;
   WHILE v <> temp2 DO
      BEGIN
      v := PRED(v);
      body
      END
   END
END
```

Examples of FOR statements:

```
FOR I := 2 TO 63 DO
   IF A[I] > MAX THEN MAX := A[I]

FOR C := RED TO BLUE DO WRITELN(C)

FOR J := K DOWNTO 1 DO SUM := SUM + J
```

Conditional Statements

A conditional statement selects one of a number of alternate courses of action based upon the evaluation of a certain condition. There are two types of conditional statements -- IF and CASE.

IF Statement: The form of an IF statement can be either:

```
IF Boolean-expression THEN statement-1
```

or

```
IF Boolean-expression THEN statement-1
   ELSE statement-2
```

where statement-1 and statement-2 may be any statement, including a compound statement.

When the IF-THEN form is used, statement-1 is executed only if the Boolean-expression is true. Otherwise, statement-1 is bypassed and the next sequential statement is executed.

The IF-THEN-ELSE form allows the selection of one of two statements depending upon the value of the Boolean-expression. If the Boolean-expression is true, statement-1 is executed and statement-2 is bypassed. If the Boolean-expression is false, statement-1 is bypassed and statement-2 is executed.

After the execution of the IF statement, control is passed to the next sequential statement. However, if the executed statement-1 or statement-2 contains a GOTO statement, then control is transferred according to the rules for the GOTO statement.

Examples:

```
IF A > B THEN WRITELN(' A IS GREATER THAN B.')

IF x < 1.5
    THEN z := x + y ELSE z := 1.5   {There must never be a
                                     semicolon before ELSE,
                                     because ELSE is not a
                                     statement but part of one.}
```

An IF statement is said to be nested whenever it appears as statement-1 or statement-2 or as part of statement-1 or statement-2. In these cases, any ELSE encountered must be paired with the immediately preceding IF which has not been already paired with an ELSE. The number of ELSEs in a nested IF structure need not be the same as the number of IFs. Examples:

```
IF X > 0 THEN
   IF Y > 0 THEN Y := Y + 1 ELSE X := X + 1

IF A < C THEN
   IF C < D THEN X := 1 ELSE
      IF A < C THEN
         IF B < D THEN X := 2 ELSE X := 3
      ELSE
         IF A < D THEN
            IF B < C THEN X := 4 ELSE X := 5
         ELSE X := 6
ELSE X := 7
```

CASE Statement: A CASE statement is used to select one of a set of statements for execution depending upon the value of an expression. The general form of a CASE statement is:

```
CASE expression OF
   case-constant-list-1 : statement-1;
                .
                .
                .
   case-constant-list-n : statement-n
   [; OTHERWISE statement]
END
```

The expression is a case selector and can be of any scalar type, except REAL. Each case-constant-list contains one or more distinct constants of the same type as the case selector. Each of these constants indicates one of the values of the case selector for which the associated statement is to be executed. Multiple constants in a list are separated by commas. The case-constant-list clauses may be written in any order.

Each statement controlled by a CASE statement may be any statement, including a compound statement.

Example 1:

```
VAR
   OPERATOR : (PLUS,MINUS,TIMES);
        .
        .
        .
BEGIN
        .
        .
        .
   CASE OPERATOR OF
      PLUS :  X := X + Y;
      MINUS:  X := X - Y;
      TIMES:  X := X * Y
   END;
        .
        .
        .
END.
```

Example 2:

```
TYPE
   DAYS = (SUN,MON,TUE,WED,THUR,FRI,SAT);...
VAR
   TODAY,TOMORROW,YESTERDAY : DAYS;...
BEGIN
        .
        .
```

```
        .
    FOR TODAY := SUN TO SAT DO
      BEGIN
        CASE TODAY OF
          SUN : BEGIN YESTERDAY := SAT ; TOMORROW := MON END;
          SAT : BEGIN YESTERDAY := FRI ; TOMORROW := SUN END;
          MON,TUE,WED,THUR,FRI:
                BEGIN
                  YESTERDAY := PRED(TODAY);
                  TOMORROW  := SUCC(TODAY)
                END;
        END;  {CASE statement}
      WRITELN ('TODAY', ORD(TODAY), 'TOMORROW', ORD(TOMORROW),
               'YESTERDAY', ORD(YESTERDAY));
      END;  {FOR statement}
        .
        .
        .
END.
```

Example 3:

```
    VAR
       DAYS : SUN..SAT; ...
    BEGIN
          .
          .
          .
       CASE DAYS OF
          SUN, SAT : ;  {Since there is no action required for
                         SUN and SAT, the space before the
                         semicolon is an empty statement
                         producing no action.}
          MON, WED, FRI : statement-1;
          TUE, THUR : statement-2
       END;
          .
          .
          .
END.
```

When the CASE statement is executed, the case selector must have one of the indicated values; otherwise, the effect of the CASE statement is undefined.

However, in situations where there is a large group of values for which one associated statement is to be executed, it would be impractical to list them all explicitly. In that case, OTHERWISE may be used to replace these values and the statement following OTHERWISE will be executed if no other statement in the case-constant-list clause is selected.

The OTHERWISE clause option is a Prime extension to standard Pascal. This clause, if present, must immediately precede the

keyword END which terminates the CASE statement. OTHERWISE is a Prime keyword. Example:

```
VAR
   I : 1..20; ...
BEGIN
      .
      .
      .
   CASE I OF
      1,20 : statement-1;
         4 : statement-2;
     5,7,9 : statement-3;
   3,11,17 : statement-4;
   OTHERWISE statement-5
   END;
      .
      .
      .
END.
```

In standard Pascal the function of the OTHERWISE clause can be achieved by the combination of the standard CASE statement and an IF statement in the form:

```
IF expression IN [set of case constants]
   THEN
      CASE expression OF
           .
           .
           .
      END
ELSE
           .
           .
           .
```

The previous example of the OTHERWISE clause may be rewritten in standard Pascal as:

```
VAR
   I : 1..20;
      .
      .
      .
   IF I IN [1,3,4,5,7,9,11,17,20]
      THEN
         CASE I OF
            1,20 : statement-1;
               4 : statement-2;
           5,7,9 : statement-3;
         3,11,17 : statement-4
         END
      ELSE statement-5
```

Note

The CASE statement is different from the CASE clause in the variant part of a record. The CASE clause is discussed in Section 6.

Unconditional Statement

GOTO Statement: A GOTO statement is an unconditional statement which transfers control to the statement designated by the label without testing or satisfying any condition. The form of the GOTO statement is:

GOTO label

The label is an unsigned integer up to 4 digits long; it must be declared in the label declaration prior to its appearance in the GOTO statement. The designated statement must be prefixed with the label followed by a colon.

There are some restrictions on the use of a GOTO statement. A GOTO statement can be used to transfer control within a block, or from an inner block to an outer block; it cannot be used to transfer control from an outer block to an inner block. In particular, a GOTO statement can be used to transfer control out of a subprogram (procedure or function), but not into one.

Example 1:

```
          .
          .
          .
LABEL 10;  {DATA} ...
PROCEDURE P1;
   LABEL 20, 30; ...
   BEGIN ...
   20:  IF s1 THEN GOTO 30;
          .
          .
          .
        GOTO 20;
   30: s5;
          .
          .
          .
   IF s7 THEN GOTO 10
   END; {P1}
BEGIN {DATA} ...
10: s9;  {A "GOTO 20" or "GOTO 30" is not
          permitted in DATA.}
          .
          .
```

Example 2:

```
{This is an invalid example.}

PROGRAM MAIN; ...
PROCEDURE P; ...
   BEGIN ...
     5 : s1
   END;  {Of procedure P}
BEGIN ...
   GOTO 5;  {Transferring control to an inner block
        .      is not permitted.}
        .
        .
END.  {Of program MAIN}
```

Note

In general, GOTO statements make a program algorithm hard to understand or follow and their use should be minimized. Therefore, a GOTO statement should be used only when it cannot be easily replaced by other available Pascal statement(s), such as compound, repetitive, conditional, or WITH statements.

WITH STATEMENT

A particular component of a record is normally accessed by using both the name of the record variable and the name of the component (a field identifier), separated by a period. (See Section 6.)

However, if a record component must be accessed many times in a section of a program unit, the WITH statement can simplify this access by indicating the record variable name only once.

The form of a WITH statement is:

```
WITH record-variable-1 [,record-variable-2]...
  DO statement
```

This form is equivalent to:

```
WITH record-variable-1 DO
  [WITH record-variable-2 DO]...statement
```

The statement may be any statement, including a compound statement. Within the statement, components of a record may be referred to only by field identifiers. For example,

```
WITH DATE DO
  IF MONTH = 12 THEN
    BEGIN
      MONTH := 1;
      YEAR := YEAR + 1
    END
  ELSE MONTH := MONTH + 1
```

is equivalent to:

```
IF DATE.MONTH = 12 THEN
  BEGIN
    DATE.MONTH := 1;
    DATE.YEAR := DATE.YEAR + 1
  END
ELSE DATE.MONTH := DATE.MONTH + 1
```

SECTION 9

PROCEDURES AND FUNCTIONS

In addition to the main program, a Pascal program may contain a number of procedures and functions that can be collectively called subprograms. In Prime Pascal, a subprogram has the following features:

- A subprogram can be at most 64K words (128 bytes) in size.
- A subprogram can be separated from the main program, or embedded within the main program or another subprogram.
- An independently written subprogram can be in any Prime supported language. If the subprogram is declared in a Pascal program using the EXTERN attribute, then the subprogram can be referenced from any point within the Pascal program.
- Before it is fully defined, a subprogram can be referenced by other subprograms within the same Pascal program. However, the referenced subprogram must have been declared using the FORWARD attribute.
- Recursive call is allowed; that is, a subprogram can call itself.

PROCEDURES

A procedure is an independent program unit that performs a set of operations. A procedure must be declared in a procedure declaration, a forward procedure declaration, or an external procedure declaration, before the procedure can be invoked by a procedure statement.

Procedure declarations are discussed below. Forward and external procedure declarations are discussed later in this section.

The external procedure declaration is a Prime extension to standard Pascal.

Procedure Declarations

A procedure declaration defines a procedure and associates it with an identifier, allowing the procedure to be activated by a procedure statement.

The form of a procedure declaration is:

PROCEDURE identifier [(formal-parameter-list)]; block;

The keyword PROCEDURE begins a procedure declaration. It is followed by an identifier which names the procedure and a list of identifiers, if any, called formal parameters. This list, enclosed in parentheses, specifies the name of each formal parameter followed by its type-identifier. Parameters are discussed later in this section.

Except in forward or external declarations, the procedure heading (described above) is immediately followed by the full definition of the procedure, called the procedure block.

A procedure block has the same general form as a program. It may contain declarations for labels, constants, types, variables, procedures, and functions and a sequence of executable statements surrounded by a BEGIN and END pair. However, it ends with a semicolon instead of a period.

A procedure never returns an explicit value. The name of a procedure must not be assigned a value and must be used only for identification purposes. Therefore, it is not allowed to specify a data type for a procedure itself.

All identifiers specified in the value formal-parameter-list of procedure declaration A and all identifiers and labels declared in the block of A are local to A. The scope of these identifiers and labels is limited to A and to all procedures and/or functions declared within A. These identifiers and labels are significant (or known) only inside their scope. (Concepts discussed here apply to function declarations as well.)

Note

> Identifiers and labels declared in the main program are global; they are significant throughout the entire program. However, if some of these identifiers and labels belong to a particular procedure (or function) but not to the program as a whole, redeclare them within that procedure (or function).

Invoking Procedures

A procedure statement, which is a procedure call, invokes (or calls) a procedure. A procedure statement has the form:

```
procedure-identifier [(actual-parameter-1 [,actual-parameter-2]...)]
```

The procedure-identifier is the name of the called procedure. When the called procedure has one or more formal parameters defined in its heading, the procedure statement must contain the corresponding actual parameter(s) along with the procedure-identifier.

Example 1:

```
        .
        .
        .
    PROCEDURE indata;...BEGIN...END;
    PROCEDURE sort;...BEGIN...END;
    PROCEDURE outdata;...BEGIN...END;
    {Main program begins here.}
    BEGIN
       indata;
       sort;
       outdata
    END.
```

Example 2:

```
    PROGRAM CURVE(INPUT,OUTPUT);
    VAR X,Y: REAL;
        I: INTEGER;
       .
       .
       .
    PROCEDURE PLOT(A,B: REAL; J: INTEGER); {A,B and J are formal
       .                                    parameters.}
       .
       .
    BEGIN...END;
    PROCEDURE ENDPLOT;
       .
       .
       .
    BEGIN...END;
    {Main program begins here.}
    BEGIN
      X := 0.0;
      Y := 1.6 + SIN(X);
      READLN(I);
      I := I + 2;
      PLOT(X, Y, I);  {X, Y and I are actual parameters.}
      .
      .
      .
      ENDPLOT;
      .
      .
      .
    END.
```

Standard Procedures

A standard procedure, denoted by a predefined identifier, is a built-in procedure supplied by the Pascal language.

Prime Pascal supports the following standard procedures:

- File Handling Procedures: RESET, GET, REWRITE, PUT, READ, READLN, WRITE, and WRITELN (See Section 10.)
- I/O Auxiliary Procedures: PAGE and CLOSE (See Section 10.)
- Dynamic Allocation Procedures: NEW and DISPOSE (See Section 6.)

Note

The CLOSE procedure is a Prime extension to standard Pascal.

Prime Pascal does not yet support standard transfer procedures -- PACK and UNPACK.

PARAMETERS

Parameters allow information to be passed between the calling program units and the called program units. There are two kinds of parameters -- actual and formal.

Actual Parameters

An actual parameter, appearing in a procedure statement or function designator, is a variable whose value is to be passed to the formal parameter in the corresponding position in the called procedure or function heading. The actual parameters must agree in order, number and data type but not necessarily in name, with the formal parameters.

Formal Parameters

Formal parameters are defined in the formal-parameter-list of a procedure or function heading. This list specifies the order, number and data type of the corresponding actual parameters.

The general form of a formal-parameter-list is:

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

The identifier-list contains one or more identifiers (separated by

commas) of the same type denoted by the type-identifier. Identifiers specify the names of the variables used in the procedure or function.

In standard Pascal, there are four kinds of formal parameters -- value, variable, procedure, and function. Prime Pascal only supports value parameters and variable parameters.

Value Parameters: If an identifier-list in a formal-parameter-list is not preceded by the keyword VAR, then its components are value parameters.

A value parameter denotes a distinct, local variable in the called procedure or function. The corresponding actual parameter is an expression whose value must be assignment-compatible with the type of the value parameter. When the subprogram is called, the current value of the actual parameter (expression) is assigned to the variable. The subprogram may then assign other values to this variable. However, the values assigned in the subprogram do not change the value of the actual parameter in the calling program unit. Therefore, when the values of actual parameters need to be protected, use value parameters. Value parameters can be regarded as pass-by-value parameters. Example:

```
PROGRAM PARAMETERS;
VAR A,B : INTEGER;
PROCEDURE VALUE_PAR(I, J : INTEGER);
  BEGIN
    I := I + 1;  {I = 2}
    J := J + 2;  {J = 3}
  END;  {Procedure VALUE_PAR}
BEGIN
  A := 1; B := 1;
  VALUE_PAR(A,B);
  WRITELN(A,B);  {A=1, B=1}
END.
```

Variable Parameters: If an identifier-list in a formal-parameter-list is preceded by the keyword VAR, then its components are variable parameters.

A variable parameter denotes the corresponding actual parameter during the entire activation of the procedure or function. Any operation involving the variable parameter is performed immediately on the actual parameter, that is, changes to the variable parameter will cause corresponding changes to the actual parameter. A variable parameter can be regarded as a pass-by-reference parameter and its corresponding actual parameter can only be a variable. The variable parameter and the actual parameter must be of the same type. Example:

```
PROGRAM PARAMETERS;
VAR A,B : INTEGER;
PROCEDURE BAR(VAR I : INTEGER; J : INTEGER);
                        {I is a variable parameter;
                         J is a value parameter.}
  BEGIN
    I := I + 5;  {I = 7}
    J := J + 3;  {J = 4}
  END;  {Procedure BAR}
BEGIN
  A := 1;
  B := 1;
  BAR(A, B);
  WRITELN(A, B)  {A = 7; B = 1}
END.
```

Caution

Do not pass a constant to a variable parameter, or a compile time error will be generated. The following example is invalid:

```
PROGRAM MAIN; ...
PROCEDURE FOO(VAR X : INTEGER); ...
  BEGIN ... END;
BEGIN
    .
    .
    .
  FOO(10);
    .
    .
    .
END.
```

ARRAY or RECORD Type Formal Parameters

Prime Pascal compiler can produce two types of object code. Ordinary code can address only within a segment. Boundary-spanning code is capable of addressing across the boundary between one segment and the next.

Whenever an array or record extends across a segment boundary, all references to it must consist of boundary-spanning code, because those portions of it in segments higher than the one in which it begins are inaccessible to ordinary code.

All references in the program to any array or record the compiler knows to be longer than a segment will automatically be compiled with boundary-spanning code. No special action is required of the user in this case.

However, when an ARRAY or RECORD type formal parameter occurs in a

subprogram, the compiler has no way to know the storage status of any ARRAY or RECORD type actual parameter that will become associated with it when the subprogram is invoked. Therefore, the compiler cannot know whether to compile references to that formal parameter with ordinary or boundary-spanning code. It is the user's responsibility to inform the compiler of the correct action in this case, through use of the -BIG/nobig compiler option.

When a subprogram is compiled without -BIG (nobig is the default), ARRAY or RECORD type formal parameter references within it will be compiled with ordinary code; the corresponding ARRAY or RECORD type actual parameter must then be contained within one segment.

When a subprogram is compiled with -BIG, all references it makes to any ARRAY or RECORD type formal parameter will be compiled with boundary-spanning code; the corresponding ARRAY or RECORD type actual parameter may then span a segment boundary, though it need not do so.

An ARRAY or RECORD type formal parameter reference compiled with boundary-spanning code will execute correctly for any ARRAY or RECORD type actual parameter, whether it spans a segment boundary or not. However, boundary-spanning code executes more slowly than ordinary code because it performs more complex address calculation. The -BIG option should therefore not be used unnecessarily.

FUNCTIONS

A function is an independent program unit which accepts one or more input values to produce a single output value. A function must be declared in a function declaration, a forward function declaration, or an external function declaration, before the function can be invoked by the evaluation of a function designator.

Function declarations are discussed below. Forward and external function declarations are discussed later in this section.

The external function declaration is a Prime extension to standard Pascal.

Function Declarations

A function declaration defines a function which computes a value of a scalar type or a pointer type. The function declaration is similar to a procedure declaration with the following exceptions:

- The keyword FUNCTION is used instead of PROCEDURE.
- A data type for the function itself is specified in its heading since a function always returns a value.

- The result value is assigned to the function name itself within the associated block.

The general form of a function declaration is:

```
FUNCTION identifier [[(formal-parameter-list)]:
    result-type-identifier]; block;
```

The result-type-identifier denotes either a scalar or a pointer type. (Data types are explained in Section 6.)

Example 1:

```
FUNCTION sqrt(X : REAL) : REAL;
{This function computes the square root of X (X>0) using Newton's
method.}
VAR OLD, NEW : REAL;
BEGIN
  NEW := X;
  REPEAT
    OLD := NEW;
    NEW := (OLD + X/OLD) * 0.5;
  UNTIL ABS(NEW - OLD) < EPS * NEW; {EPS being a global constant}
  sqrt := NEW
END;  {Function Sqrt}
```

Example 2:

```
FUNCTION MAX(A : VECTOR; N : INDEXTYPE) : REAL;
{This function finds the largest value in A, which is declared
  A : ARRAY[INDEXTYPE] OF REAL and where
  INDEXTYPE = 1..LIMIT}
VAR
  LARGESTSOFAR : REAL;
  FENCE : INDEXTYPE;
BEGIN
  LARGESTSOFAR := A[1];
  {Establishes LARGESTSOFAR = MAX(A[1])}
  FOR FENCE := 2 TO N DO
    BEGIN
      IF LARGESTSOFAR < A[I]
        THEN LARGESTSOFAR := A[I]
        {Re-establishing LARGESTSOFAR = MAX(A[1],...,A[FENCE])}
    END;
  {So now LARGESTSOFAR = MAX(A[1],...,A[N])}
  MAX := LARGESTSOFAR
END; {Function MAX}
```

Invoking Functions

The evaluation of a function designator invokes a function. A function designator is a component of an expression: that is, it can only appear on the right-hand side of an assignment statement. A function designator has the form:

```
function-identifier [(actual-parameter-1 [, actual-parameter-2]...)]
```

The function-identifier is the name of the called function. When the called function has one or more formal parameters defined in its heading, the function designator must contain the corresponding actual parameter(s) along with the function-identifier. An actual-parameter can be an expression or a variable. Example:

```
VAR J,K : INTEGER;
FUNCTION CUBE(I : INTEGER) : INTEGER;
  BEGIN
    CUBE := I * SQR(I);
  END; {Function CUBE}
BEGIN
  READLN(J);
  K := CUBE(J);  {Function CUBE is invoked here.}
  .
  .
  .
END.
```

Standard Functions

A standard function, denoted by a predefined identifier, is a built-in function supplied by the Pascal language. The available standard functions are listed and explained in Section 11.

FORWARD PROCEDURES AND FUNCTIONS

Pascal permits subprograms to call each other within the same Pascal program. Subprogram A may call subprogram B before B is fully defined if B has already been declared using the forward declaration.

Forward Declarations

A forward declaration is declared like other subprogram declarations, except that the subprogram block is replaced by the word FORWARD. This block, led by the keyword PROCEDURE or FUNCTION and its associated subprogram name, appears later in the program. Example:

```
FUNCTION GCD(N,M : INTEGER) : INTEGER; FORWARD;
PROCEDURE LOWTERM(VAR L,D : INTEGER);
VAR
```

```
  CD : INTEGER;
BEGIN
  CD := GCD(N,D);  {This statement needs the forward declaration.}
  N := N DIV CD;
  D := D DIV CD
END; {Procedure LOWTERM}
FUNCTION GCD;  {Note the abbreviated heading}
{Full declaration of GCD begins here.}
VAR
  R : INTEGER;
BEGIN
  REPEAT
    R := M MOD N;
    IF R <> 0 THEN
      BEGIN
        M := N;
        N := R
      END;
  UNTIL R = 0;
  GCD := N
END; {Function GCD}
```

EXTERNAL PROCEDURES AND FUNCTIONS

Prime Pascal allows a program to call external, separately compiled subprograms after they have been declared using the external declarations within the program.

External Declarations

An external procedure or function declaration is similar to a forward procedure or function declaration with the word FORWARD replaced by EXTERN. The body (block) of an external subprogram does not appear in the program. It will be located and included automatically when the program is loaded. Example:

```
PROGRAM CIRC(INPUT,OUTPUT);
  .
  .
  .
PROCEDURE PLOT(X,Y : REAL; IPEN : INTEGER); EXTERN;
{The main program starts here.}
BEGIN
  .
  .
  .
PLOT(X,Y,3);
  .
  .
  .
PLOT(X,Y,2);
```

```
    .
    .
    .
END.
```

Subprograms Written in Pascal

When an external subprogram is written in Pascal, separate compilation of the subprogram can be achieved by including the {$E+} or (*$E+*) compiler switch at the beginning of the module (text that is input to the compiler). The presence of this compiler switch causes the following to take place:

- All procedures and functions declared at the program level (outermost level) become globally defined entry points that can be referenced by other Pascal programs which have declared them using the EXTERN attribute.
- All variables, constants, and types declared at the program level (outermost level) are considered external variable, constant, and type definitions (as in the PL/I external attribute) and are available to all procedures or functions loaded with the module.
- The normally required program "BEGIN END." pair is no longer required.

Note

SEG limits the length of external names to 8 characters. Therefore, all external constant, type, variable, procedure, and function names should be at most 8 characters in length.

Compiler switches are discussed in detail in Section 2.

Example:

```
{$E+ Enable external compilation}

CONST
  MAXRANGE = 50;     {Global constant definition}

TYPE
  STRING10 = ARRAY[1..10] OF CHAR;
                     {Global variable definition}

VAR
  GLOBALS : RECORD  {Global variable definition}
              DATA : ARRAY[1..MAXRANGE] OF REAL;
              MOREDATA : CHAR
            END;
```

```
PROCEDURE MODULE_1(VAR P1, P2: INTEGER);
  VAR
    S1 : STRING[10];  {VAR defined using global type}

  BEGIN
  S1[1] := GLOBALS.MCREDATA;  {<local> := <global>}
    .
    .
    .
  END;  {End of MODULE_1}

PROCEDURE MODULE_2(VAR P1, P2, P3: REAL);
    .
    .
    .
```

Caution

Do not define a program as external, or an access violation will result at run-time. The following example is invalid:

```
{$E+}
PROGRAM FOO(INPUT, OUTPUT);
  .
  .
  .
```

Subprograms Written in Other Languages

Subprograms declared in external procedure or function declarations can be written in any Prime high-level language or Prime Macro Assembly language with certain restrictions:

- There must be no conflict of data types for variables being passed as parameters. For example, a FIXED BINARY(15) in PL/I should be declared as INTEGER in Pascal.
- Modules compiled in either 64V or 32I mode cannot reference or be referenced by modules compiled in any R mode. Modules in 64V or 32I may reference each other if they are otherwise compatible.

Subprograms from Libraries

Prime supplies several libraries of application-level subroutines and PRIMOS operating system subroutines. These subroutines can be declared in external procedure or function declarations of a Pascal program, then referenced from any point within the program. When a subroutine from such a library has been referenced, the command:

```
LI library-name
```

must be given to SFO at load time before the unqualified LI command is given.

For more information, see The PRIMOS Subroutines Reference Guide.

SECTION 13

INPUT AND OUTPUT

Input/Output is performed in Pascal by the proper combined use of two Boolean functions EOF and EOLN, eight file handling procedures RESET, GET, REWRITE, PUT, READ, READLN, WRITE and WRITELN, and two auxiliary procedures PAGE and CLOSE.

Throughout this section, file and file-1 used in the parameter list of each function or procedure are the file variables. When the file variables are the predefined file variables INPUT and OUTPUT, they provide I/O to and from the user's terminal.

EOF FUNCTION

The EOF function tests for an end-of-file condition of an input file. The form of EOF is:

```
EOF(file)
```

This function is true if the buffer variable file^ has moved beyond the end of the file. (Buffer variable is discussed in Section 6.) Otherwise it is false. The parameter file is of a FILE type. If file is omitted, EOF is applied to the standard textfile INPUT. That is, EOF is equivalent to EOF(INPUT).

RESET PROCEDURE

All input files except INPUT must be reset by using the RESET procedure before reading.

A RESET procedure has the form:

```
RESET(file-1, file-2)
```

where the first parameter file-1 is a Pascal file variable and the second parameter file-2 denotes the name of an actual file to be opened for input. The name can be either a filename if the actual file resides in the current UFD or a pathname if the actual file does not reside in the current UFD. The second parameter file-2 is a Prime extension to standard Pascal.

RESET associates file-2 with file-1, resets the current file position of file-1 to its beginning and assigns the value of the first element of file-1 to the buffer variable file-1^. EOF(file-1) is false, if file-1 is not empty.

Once the association is established by a call to the RESET procedure, the same file-2 is implied in repeated resetting of file-1 if the second parameters in these RESET procedures are not specified.

Example 1:

```
BEGIN  . . .
RESET(X, 'REALNAME');  {Assume that REALNAME is the actual
      .                 input filename.  It must be specified
      .                 in the first call of a RESET
      .                 procedure.}
RESET(X);  {X is still associated with the same input file
      .     REALNAME.}
      .
      .
RESET(X, 'REALNAME2');  {REALNAME is closed and X is now
      .                  associated with another input file
      .                  REALNAME2.}
      .
CLOSE(X);  . . .
END.
```

Example 2:

```
BEGIN  . . .
RESET(Fyle, 'GRACE>Fyle1');
          .
          .
          .
RESET(Fyle, 'GRACE>Fyle1');  {This call is equivalent to
          .                   RESET(Fyle).}
          .
          .
          .
CLOSE(Fyle);
          .
          .
          .
RESET(Fyle, 'GRACE>Fyle1'); . . . CLOSE(Fyle);  . . .
END.
```

file-1 is of a FILE type. file-2 may be either a pathname or filename enclosed in a pair of apostrophes (as shown in the above examples) or a variable of a string type.

If a pathname or filename is used to denote file-2, it is specified only in the RESET procedure.

Example 1:

```
VAR
  X : FILE OF CHAR;  . . .
BEGIN
  RESET(X, 'C');  . . .
```

Example 2:

```
TYPE
  IOREC = RECORD
            A: INTEGER;
            B: ARRAY [1..6] OF CHAR;
            C: (LEFT, RIGHT)
          END;
VAR
  F: FILE OF IOREC;...
BEGIN
  RESET (F, 'F1');...
```

If a variable is used to represent file-2, the variable and its associated string type must first be declared in a variable declaration:

identifier : ARRAY [1..n] OF CHAR;

where identifier is the name of the variable. n specifies the length of the array (pathname or filename). Example:

A : ARRAY [1..4] OF CHAR;

The pathname or filename must then be enclosed in a pair of apostrophes and assigned to the variable in an assignment statement. Example:

A := 'TEST';

Once the assignment is done, file-2 can be specified either by the variable name on the left-hand side of the assignment operator or by the pathname or filename on the right-hand side of the operator. Example:

```
VAR
  INFILE : FILE OF INTEGER;
  A : ARRAY [1..4] OF CHAR;
        .
        .
        .
BEGIN
  A := 'TEST';
  RESET(INFILE, A);  {This call is equivalent to
        .                RESET(INFILE, 'TEST').}
        .
        .
```

GET PROCEDURE

After the RESET procedure opens and resets an input file, the GET procedure reads the next element of the file.

The form of GET is:

```
GET(file)
```

GET advances the current file position to the next element, assigns the value of this element to the buffer variable file^, and leaves EOF(file) false. If no next element exists, EOF(file) becomes true and file^ is not defined. The parameter file is of a FILE type.

REWRITE PROCEDURE

All output files except the standard textfile OUTPUT must be opened by using the REWRITE procedure before writing.

A REWRITE procedure has the form:

```
REWRITE(file-1 [, file-2])
```

The first parameter file-1 is a Pascal file variable of a FILE type.

The optional second parameter file-2 denotes the filename or pathname of an actual file to be opened for output. If file-2 is not specified in the first call of a REWRITE procedure, a filename T$xxxx will be generated automatically for the output file, where xxxx is a 4-digit number. If file-2 is specified, it may be either a filename or pathname enclosed in a pair of apostrophes or a variable of a string type representing the filename or pathname. The second parameter option is a Prime extension to standard Pascal.

REWRITE associates the actual output file with file-1, empties file-1 completely, sets EOF(file-1) to true, and initializes file-1 for writing.

Once the association is established by the REWRITE procedure, the same file-2 or T$xxxx is implied in repeated rewriting of file-1 if the second parameters in these REWRITE procedures are not specified.

Example 1:

```
VAR OUTDATA: FILE OF ARRAY [1..7] OF CHAR; ...
BEGIN
    .
    .
    .
  REWRITE (OUTDATA);  {OUTDATA will be associated with an output
    .                   file T$xxxx.}
    .
    .
  REWRITE (OUTDATA, '<IDR4303>EXAMPLE'):  {T$xxxx is closed and
    .                                       OUTDATA is now
    .                                       associated with
    .                                       another output file
```

```
      .                                          EXAMPLE.)
    REWRITE (OUTDATA);  {OUTDATA is still associated with the same
      .                  output file EXAMPLE.}
      .
      .
  END.
```

Example 2:

```
  VAR
    OUTFILE: TEXT;
    A:  ARRAY [1..12] OF CHAR;  {The pathname or filename of an
        .                       output file is 12-character long.}
        .
        .
  BEGIN
    A:= '&RACE>SAMPLE';
    REWRITE (OUTFILE, A);  {This call is equivalent to
        .                   REWRITE (OUTFILE, '&RACE>SAMPLE').}
        .
        .
  END.
```

PUT PROCEDURE

After the REWRITE procedure opens an output file, the PUT procedure writes the output value into the file.

The form of PUT is:

```
PUT(file)
```

PUT appends and writes the value of the buffer variable file^ to the end of the file. EOF(file) remains true. The file is of a FILE type.

The use of EOF, RESET, GET, REWRITE and PUT is illustrated in the following example:

```
VAR INFILE, OUTFILE: TEXT;
BEGIN
  RESET(INFILE, 'INDATA');
  REWRITE(OUTFILE, 'OUTDATA');
  WHILE NOT EOF(INFILE) DO
    BEGIN
      OUTFILE^ := INFILE^;
      PUT(OUTFILE);
      GET(INFILE)
    END;
  CLOSE(INFILE);  {The CLOSE procedure is discussed at the end
  CLOSE(OUTFILE)   of this section.}
END.
```

EOLN FUNCTION

The function EOLN tests for an end-of-line condition in an input textfile. It has the form:

```
EOLN(file)
```

This function is true if the buffer variable file^, which is currently a blank, corresponds to the position of a line separator marking the end of the current line. The line separator can be one of two ASCII characters CR (carriage return) or LF (line feed). EOLN is applied to the standard textfile INPUT if the parameter file is omitted.

READ PROCEDURE

The READ procedure reads input data values and assigns these values, by position, to the variables in the READ parameter list. It has the form:

```
READ ([file,] variable-1 [,variable-2]...)
```

The file must be of type TEXT (that is, FILE OF CHAR), the variables can be of type CHAR (or a subrange of type CHAR), INTEGER (or a subrange of type INTEGER), or REAL.

When the variable is of type CHAR (or a subrange thereof), the call:

```
READ(file, variable)
```

is equivalent to:

```
BEGIN
  variable := file^;
  GET(file)
END
```

When the variable is of type INTEGER (or a subrange thereof) or REAL, the call:

```
READ(file, variable)
```

reads a sequence of characters that forms an integer or a real number according to the rules for NUMERIC CONSTANTS (see Section 4) and assigns the number to a variable. Successive numbers are separated by blanks or ends of lines.

A READ procedure may have several parameters. The call:

```
READ (file, variable-1,...,variable-n)
```

is equivalent to:

```
BEGIN
  READ (file, variable-1);
          .
          .
          .
  READ (file, variable-n)
END
```

The first parameter file may be omitted. If it is omitted, the READ procedure is applied to the standard textfile INPUT.

READLN PROCEDURE

The READLN procedure is a variant of the READ procedure. It has the form:

```
READLN[([file|variable-1][,variable-2]...)]
```

READLN must only be applied to textfiles. If the first parameter file is omitted, the standard textfile INPUT is assumed.

READLN skips over characters until the end of the current line and places the current file position at the beginning of the next line. Thus, the call:

```
READLN(file)
```

is equivalent to:

```
BEGIN
  WHILE NOT EOLN(file) DO GET(file);
  GET(file)
END
```

READLN may also read the input data values into variables and then skip to the beginning of the next line. The call:

```
READLN(file, variable-1,...,variable-n)
```

is equivalent to:

```
BEGIN
  READ(file, variable-1,...,variable-n);
  READLN(file)
END
```

WRITE PROCEDURE

The WRITE procedure writes the values of the text and/or expressions into the output textfile. It has the general form:

```
WRITE([file,] write-parameter-1 [,write-parameter-2]...)
```

The file must be of type TEXT (that is, FILE OF CHAR). If file is omitted, the WRITE procedure is applied to the standard textfile OUTPUT. If the WRITE procedure has several parameters, the call:

```
WRITE(file, write-parameter-1,...,write-parameter-n)
```

is equivalent to

```
BEGIN
  WRITE (file, write-parameter-1);
     .
     .
     .
  WRITE (file, write-parameter-n)
END
```

Write-parameters

A write-parameter is either a character string enclosed by a pair of apostrophes or an expression (represented by its variable name) with optional field-width parameter(s).

If the write-parameter is a character string, it is written on the output file exactly as it appears, without the delimiting apostrophe characters. For example:

```
WRITE('The ''blank character'' is significant in a character-string.')
```

will produce:

```
The 'blank character' is significant in a character-string.
```

If the write-parameter is an expression, the value of the expression is written in the output file. This write-parameter has the form:

```
expression [: total-width [: frac-digits]]
```

The expression may be of type INTEGER, REAL, CHAR, or BOOLEAN. The total-width and frac-digits are field-width parameters; they may only be of type INTEGER.

total-width is the total number of character positions to be allocated

for the value of the expression. If it is omitted, a default field-width will be assumed according to the type of the expression. The default field-widths are as follows:

Data Type	Number of Character Positions
INTEGER	10
REAL	22
CHAR	1
BOOLEAN	10

A real value for which no frac-digits are specified will be written in floating-point (or scientific) form:

d...d.ddddddE+|-cc

6-digit

where each d denotes a decimal digit. If the default field-width is used, there should be 11 digits preceding the decimal point. Otherwise, this number of digits should depend on the total-width specified. For example, if R and S are real variables with values 0.1 and 1.5 respectively, the WRITE procedure:

```
WRITE(R, S:14)
```

will produce the following:

```
bbbbbbbbbb1.000000E-01    bb1.500000E+00
(Default case)
```

where each b is a blank.

frac-digits, if present, can only be applied to an expression of type REAL. It invokes fixed-point (or decimal) representation for a real value and specifies the number of digits following the decimal point. For example, if R and S are real variables with values 1.5 and 112.123 respectively, the WRITE procedure:

```
WRITE(R:7:2, S:10:1)
```

will produce the following:

```
bbb1.50     bbbbb112.1
```

where each b is a blank.

The previous examples also show that if the true field-width of the expression is smaller than the default field-width or the total-width (if present), the unused character positions at the left are filled with blanks.

If the true field-width is larger, Pascal will automatically extend the default field-width or the total-width (if specified) to a sufficient size. For example, if B is a boolean variable with value FALSE and R is a real variable with value -2112.125, the WRITE procedure:

```
WRITE(B:1, R:3:2)
```

will produce the following:

```
FALSE     -2112.12
```

A positive real number must be written with at least one preceding blank; this, however, does not apply to values of other types. For example, if R, S, and T are real variables with values 1.5, +1.5, and -1.5 respectively, B is a boolean variable with value TRUE, and I is an integer variable with value 6, the WRITE procedure:

```
WRITE(R:7:5, S:7:5, T:7:4, B:4, I:1)
```

will produce the following:

```
b1.50000  b1.50000  -1.5000  TRUE   6
```

where b is a blank.

If the expression is of type CHAR, the call:

```
WRITE(file, expression)
```

is equivalent to:

```
BEGIN
  file^ := expression;
  PUT(file)
END
```

WRITELN PROCEDURE

The WRITELN procedure is a variant of the WRITE procedure. It has the form:

```
WRITELN[([file|write-parameter-1][,write-parameter-2]...)]
```

For the description of write-parameters, see the preceding write-parameters section.

WRITELN must only be applied to textfiles. If the first parameter file is omitted, the standard textfile OUTPUT is assumed.

WRITELN writes the values into the current line of the output file, then sends a carriage return to the file. Thus, the call:

```
WRITELN(file,write-parameter-1,...,write-parameter-n)
```

is equivalent to:

```
BEGIN
  WRITE(file,write-parameter-1,...,write-parameter-n);
  WRITELN(file)
END
```

If a WRITELN procedure is called with a single parameter file or no parameters at all, WRITELN simply sends a carriage return to the output file.

PAGE PROCEDURE

The form of the PAGE procedure is:

```
PAGE(file)
```

The PAGE procedure generates a skip to the top of a new page before the next line of the textfile file is written. If the single parameter file is omitted, this procedure is applied to the standard textfile OUTPUT.

Example:

```
BEGIN
  WRITELN('Page Test');
  WRITELN('Page 1');
  PAGE;
  WRITELN('Page 2')
END.
```

CLOSE PROCEDURE

All files (except the standard textfiles INPUT and OUTPUT) must be explicitly closed using the CLOSE procedure. Otherwise they will remain open after the program terminates.

The form of the CLOSE procedure is:

```
CLOSE(file)
```

The CLOSE procedure is a Prime extension to standard Pascal.

Example:

```
VAR
  Fyle: TEXT;
BEGIN
  REWRITE(Fyle, 'FYLE');
  WRITELN(Fyle, 'ABC');
  WRITELN(Fyle, 'DEF');
  CLOSE(Fyle)
END.
```

SECTION 11

STANDARD FUNCTIONS

A standard function, denoted by a standard identifier, is a built-in function supplied by the Pascal language. There are four types of standard functions -- arithmetic, transfer, ordinal, and Boolean.

ARITHMETIC FUNCTIONS

ABS(x) Computes the absolute value of x. The type of x must be either INTEGER or REAL. The type of the result is the same as that of x.

SQR(x) Computes the square of x. x and the result must be of the same data type, INTEGER or REAL.

Note

For the following arithmetic functions, the type of x must be either INTEGER or REAL. The type of result is always REAL.

SIN(x) Computes the sine of x.

COS(x) Computes the cosine of x.

EXP(x) Computes the value of the base of natural logarithms raised to the power x. This is exponential function (e).

LN(x) Computes the natural logarithms of x. x must be greater than zero.

SQRT(x) Computes the non-negative square root of x. x must be non-negative.

ARCTAN(x) Computes the principal value, in radians, of the arctangent of x.

TRANSFER FUNCTIONS

TRUNC(x) x must be of type REAL. The result is of type INTEGER. If x is positive then the result is the greatest integer less than or equal to x; otherwise it is the least integer greater than or equal to x. Examples:

TRUNC(3.7) yields 3
TRUNC(-3.7) yields -3

ROUND(x) x must be of type REAL. The result is of type INTEGER; it is the value x rounded. That is, if x is positive, ROUND(x) is equivalent to TRUNC (x + 0.5); otherwise ROUND(x) is equivalent to TRUNC (x - 0.5). Examples:

ROUND(3.7) yields 4
ROUND(-3.7) yields -4
ROUND(3.2) yields 3
ROUND(-3.2) yields -3

ORDINAL FUNCTIONS

ORD(x) x is an argument of any scalar type (except REAL). The result is an integer value which is the ordinal number of the argument x. If there is no such value, an unpredictable result will occur. Examples:

```
ORD('J') = 260  {Use the ASCII character set.}
ORD(plus) = 0   {Assume:
ORD(minus) = 1     VAR
ORD(times) = 2       operator:(plus,minus,times)}
```

CHR(x) x must be of type INTEGER. The result is a character whose ordinal number is x. If ch is any value of type CHAR and x is any value of type INTEGER, the following are true:

CHR(ORD(ch)) = ch
ORD(CHR(x)) = x

SUCC(x) x is of any scalar type (except REAL). The result is a value whose ordinal number is one greater than that of x. If there is no such value, an unpredictable result will occur. x and the result must be of the same type. If ch is any character, SUCC(ch) is equivalent to CHR(ORD(ch)+1).

PRED(x) x is of any scalar type (except REAL). The result is a value whose ordinal number is one less than that of x. If there is no such value, an unpredictable result will occur. x and the result must be of the same type. If ch is any character, PRED(ch) is equivalent to CHR(ORD(ch)-1).

BOOLEAN FUNCTIONS

ODD(x) — x must be of type INTEGER. The result is true if x is odd and false otherwise.

EOF(f) — f is the file variable of an input file. This function returns the value true if an end-of-file condition exists for f and false otherwise. It applies to the standard textfile INPUT if the argument f is omitted.

EOLN(f) — f is the file variable of an input textfile. This function returns the value true if the end of the current line is reached and false otherwise. It applies to the standard textfile INPUT if f is omitted.

APPENDIX A

ERROR MESSAGES

TYPES OF PASCAL ERROR MESSAGES

This appendix contains the following two categories of errors:

- Compile-time error messages
- Run-time error messages

Error messages appear alphabetically within each category. Most of these error messages are self-explanatory, otherwise they are given additional explanations.

COMPILE-TIME ERROR MESSAGES

";" not allowed between THEN and ELSE parts

"=" used instead of ":=" in assignment statement

"END" missing at end of record definition

Record definition must always be closed with the keyword END.

%INCLUDE must be followed by a quoted string containing the filename to be copied

The syntax of %INCLUDE is:

```
%INCLUDE 'filename';
```

%INCLUDE statements nested too deep

%INCLUDE can only be nested 7 levels.

Array indices can only be non-real scalar types

Index-types of arrays can only be CHAR, INTEGER subranges, or enumerated types.

Bad constant

Bad data type in variable

Bad function argument type

Bad function name

Bad name in scalar type definition

Bad operand in expression

Bad parameter name

Bad procedure name

Bad program name -- must not be a reserved word

Bad record field name

Bad type definition

Bad variable name

Bad variant tag

Base type of set must be of non-real scalar type

Comma used instead of semicolon

Constant subscript out of the bounds for this array

Decimal point a numeric literal must be followed by a digit

Real numbers must be of the form:

digit(s).digit(s)

For example: use 9.0 or 0.9 rather than 9. or .9

Don't repeat parameter list on forward definitions

Only the keyword PROCEDURE or FUNCTION with the name of the forward procedure or function should immediately precede the body (block) of that procedure or function.

DOWNTO or TO keyword is missing

Duplicate definition of label

Duplicate tag field in CASE statement

Field formatting expressions must be of type integer

Field list must be in parentheses

File variable in READ or WRITE must be FILE OF CHAR

This is a Prime restriction.

First operand must be a file variable

FOR loop index must be declared as a local variable to this block

The following example is invalid:

```
PROGRAM FX;
VAR I:  INTEGER;
PROCEDURE PRO;
  BEGIN
    FOR I := 1 TO 10 DO WRITELN(I)
  END;  {of PRO}
BEGIN
END.  {of FX}
```

Function assignment must occur within the scope of the function body

Function declared forward is never defined

The body (block) of the forward function must appear in the program.

Identifier must be declared as a pointer to use the arrow operator on it

Identifier must be declared as a record to use the dot operator on it

Identifier must be declared as an array to use subscripts on it (perhaps a missing operator or set expression)

Identifiers cannot be longer than 32 characters

Improper symbol

The following keywords must not be prefixed by any label: PROGRAM LABEL, CONST, TYPE, VAR, FUNCTION, PROCEDURE, and BEGIN (the one that designates the start of the executable part of a program block).

Incompatible types in expression

Inconsistent types of operands in set construction

Index of FOR loop must be of non-real scalar type

Invalid character

The first character of an identifier must not be an underscore _.

Invalid declaration, probably missing "END"

Invalid symbol

Label defined at a level other than it was declared at

The following example is invalid:

```
PROGRAM PX;
LABEL 100;  ...
PROCEDURE PRO;
    BEGIN
      100 : statement-1;
                 .
                 .
                 .
            GOTO 100
   END;  {of PRO}
BEGIN
```

```
END.
```

Label definitions must be unsigned integers that are not already defined in this block

Label not declared in a LABEL declaration

Label that has been declared and referenced is never defined

The following example is invalid:

```
PROGRAM TEST;
LABEL 120;
VAR I : INTEGER;
BEGIN
   READ (I);
   IF I = 0 THEN GOTO 120;
   WRITELN (I)
END.
```

Labels cannot be greater than 9999

Long integers are not allowed as array indices

Long integers are not allowed as tag fields in CASE statements

Lower limit of subrange greater than upper limit

Max nesting count exceeded

Only 64 procedures within procedures are allowed.

Missing "("

Missing ")"

Missing ")" at end of parameter list

Missing ".."

Missing ":"

Missing ":="

Missing ";"

Missing "="

Missing "BEGIN" keyword

Missing "DO"

Missing "OF"

Missing "THEN"

Missing "UNTIL" clause for REPEAT statement

Missing "["

Missing "]"

Missing colon before type identifier in declaration

Missing dot at program end

Missing keyword "LAB"

Missing label

Missing parameter for call to predefined function

Missing quote at end of string literal

Missing REPEAT statement

Must be a constant of scalar type other than real

Must be record in with statement

Only RECORD types may appear in the variable list of a WITH statement.

Non-standard Prime extension

Not a valid field name for this record type

If record Y and the record variable X are declared as follows:

```
TYPE Y = RECORD
           A, B, AB : INTEGER
         END;
VAR X : Y;
```

then only X.A, X.B, and X.AB are valid field names.

Null strings are not allowed

Number of parameters to procedure or function must be < 64

Operand to a READ statement must be of type INTEGER, REAL or CHAR

Operand to a WRITE statement must be of type INTEGER, BOOLEAN, REAL, CHAR or a string

This is a Prime restriction.

Operand to NOT operator must be boolean

Operand to unary + or - must be of type real or integer

Operands are of incompatible type for this operator

PACKED types not supported

The keyword PACKED used in type definitions does not have any effect. However, use of PACKED is not advised, and will generate a severity 1 error (warning) at compilation.

Pascal feature not implemented

Percent character only allowed in %INCLUDE statements

PROCEDURE and FUNCTION parameters not supported

Procedure contains too many statements

Procedure declared forward is never defined

The body (block) of the forward procedure must appear in the program.

Relational operators only allowed on scalars and strings

Required comma character missing

Return type for function definition must be a pointer, scalar type or a subrange

Second argument must be of string type

Source line too lon . It must be < 130 characters long

Source skipped until this point

Subrange types do not match

Subscript type does not match declared type of index of array

The ordinals of elements in a set must lie in the range 0..255

The selector expression in a case statement must be of scalar type other than real

This item in a variable definition list is already defined in this block

This symbol is not a statement starter

Too few arguments in a PROCEDURE or FUNCTION call

Too many errors on this source line

Too many parameters in a PROCEDURE or FUNCTION call

Type INTEGER is not allowed as an array index

ARRAY [INTEGER] OF base-type exceeds the capacity of this implementation.

Type name referenced in a forward way is never declared

Something was defined as a pointer to an undefined type and that type was never later defined.

Type of actual parameter does not match type of formal parameter

An actual parammeter and its corresponding value parameter must be of compatible types. An actual parameter and its corresponding variable parameter, on the other hand, must be of the same type.

Type of case label does not match type of selector expression

Type of expression must be Boolean

Type of variable to be assigned does not match type of expression it is to take as a value

Undefined symbol

Identifiers must be defined before they are used.

Variable must be of type pointer to be used as an argument to NEW or DISPOSE

RUN-TIME ERROR MESSAGES

A WRITE CANNOT BE PERFORMED TO AN UNOPENED FILE

FIELD WIDTH IS LESS THAN 1 WHICH IS NOT ALLOWED

NO NUMBER FOUND, CHECK INPUT LIST

REAL CONVERSION NOT POSSIBLE

UNABLE TO CLOSE THE FILE

UNABLE TO OPEN FILE

UNABLE TO OPEN FILE, CHECK THAT FILE EXISTS

UNABLE TO READ FROM AN UNOPENED FILE

UNABLE TO READ FROM THE BINARY FILE, CHECK DATA SIZE

UNABLE TO READ FROM THE SPECIFIED FILE, CHECK DATA SIZE

UNABLE TO REWIND FILE

This error is usually caused by a library problem.

UNABLE TO WRITE MORE THAN 256 CHARACTERS TO A TEXT FILE

UNABLE TO WRITE TO A FILE OPEN FOR INPUT

UNABLE TO WRITE TO A FILE OPEN FOR READING

UNABLE TO WRITE TO A FILE THAT IS NOT OPEN

UNABLE TO WRITE TO THE BINARY FILE, CHECK VARIABLE SIZE

UNABLE TO WRITE TO THE FILE, CHECK DATA SIZE

APPENDIX B

DATA FORMATS

OVERVIEW

The Pascal language supports the following data types:

Scalar Data Types

- INTEGER
- REAL
- CHAR
- BOOLEAN
- Enumerated
- Subrange

Structured Data Types

- ARRAY
- RECORD
- SET
- FILE

Pointer Data Type

- POINTER

These data types are described in detail in Section 6. The following shows how the data are internally represented in storage, and gives some details about each data type. In the statistics for certain data types, "P" stands for the precision (the maximum number of significant binary digits) specified when an element of this type is declared.

INTEGER TYPE DATA

A 16-bit twos-complement fixed-point binary number.

Precision: 1 <= P <= 15

Alignment: Word

Storage Requirement: 1 <= P <= 15 1 word

Internal Representation

Precision 1-15: 1 word

Bit 1: Sign
Bits 2-16: Digits

INSERT A PICTURE HERE

REAL TYPE DATA

Precision: 1 <= P <= 23

Alignment: word

Storage Requirement: 1 <= P <= 23 2 words

Internal Representation

Precision 1-23: Two words

Bit 1: Sign
Bits 2-24: Mantissa (Fraction)
Bits 25-32: Exponent

INSERT A PICTURE HERE

CHAR TYPE DATA

Length: 1 byte (parity, high order bit, always on)

Alignment: Byte-aligned

Storage Requirement: 1 byte (8-bit)

Internal Representation

One character per byte (parity always on)

INSERT A PICTURE HERE

BOOLEAN TYPE DATA

Length: 1 bit

Alignment: Word

Storage Requirement: 1 word

Internal Representation

A bit is stored in a hardware bit.

INSERT A PICTURE HERE

Note

Bit 1 on (1) is false and bit 1 off (0) is true. This representation is not compatible with FTN and F77 logical data types but is compatible with PL/I.

ENUMERATED TYPE DATA

Each ordinal number of the enumerated type is a 16-bit twos-complement fixed-point binary number. For the internal representation of an ordinal number, see INTEGER TYPE DATA of this appendix. (Ordinal numbers of an enumerated type begin at 0 and increment in a positive manner.)

SUBRANGE TYPE DATA

Refer to the BOOLEAN, CHAR, and ENUMERATED TYPE DATA for the internal representation of BOOLEAN, CHAR, and enumerated subrange types, respectively.

An INTEGER subrange constant can be either a 16- or a 32-bit twos-complement fixed-point binary number.

Precision: 1 <= P <= 15
1 <= P <= 31

Alignment: Word

Storage Requirements: 1 <= P <= 15 1 word
1 <= P <= 31 2 words

Internal Representation

Precision 1-15: 1 word

Bit 1: Sign
Bits 2-16: Digits

INSERT A PICTURE HERE

Precision 16-31: Two words

Bit 1: Sign
Bits 2-32: Digits

INSERT A PICTURE HERE

ARRAY TYPE DATA

The storage capacity and the internal representation of the ARRAY type data are determined by the index type(s) (any scalar type except REAL) and the base type (any type) specified for the elements of the array. However, elements of a string type (ARRAY[1..n] OF CHAR) will be byte alligned.

RECORD TYPE DATA

Storage of the RECORD type elements is allocated contiguously beginning with the first element. Any non-CHAR type element of a record will be word alligned while CHAR type elements will be byte alligned.

SET TYPE DATA

Length: 0 to 255 unaligned bits

Alignment: The SET type data begins on any bit by default.

Storage Requirement: 256 bits

Internal Representation

Each bit is stored in one hardware bit.

INSERT A PICTURE HERE

FILE TYPE DATA

A FILE type data item contains the address of the file control block of the indicated file.

Storage Requirement: 2 words

Internal Representation

Bit 1:	Fault code
Bits 2-3:	Ring number
Bit 4:	Data format indicator (always 0)
Bits 5-16:	Segment number
Bits 17-32:	Word number

INSERT A PICTURE HERE

POINTER TYPE DATA

Alignment: Word

Storage Requirement: 3 words

Internal Representation

Bit 1:	Fault code
Bits 2-3:	Ring number
Bit 4:	Data format indicator
Bits 5-16:	Segment number
Bits 17-32:	word number
Bits 33-36:	Bit offset (if bit 4 is set)
Bits 37-48:	Reserved

INSERT A PICTURE HERE

APPENDIX C

ASCII CHARACTER SET

The standard character set used by Prime is the ANSI, ASCII 7-bit set with the 8 parity bit always on.

PRIME USAGE

Prime hardware and software uses standard ASCII for communications with devices. The following points are particularly important to Prime usage.

- Output Parity is normally transmitted as a zero (space) unless the device requires otherwise, in which case software will compute transmitted parity. Some controllers (e.g., MLC) may have hardware to assist in parity generations.
- Input Parity is always represented as a 1 by hardware and by standard software. Input drivers are responsible for making the parity bit suit the host software requirements. Some controllers (e.g., MLC) may assist in parity error detection.
- The Prime internal standard for the parity bit is one. i.e., '200 is added to the octal value.

KEYBOARD INPUT

Non-printing characters may be entered into text using Prime's EDITOR with the logical escape character ^ and the octal value. The character is interpreted by output devices according to their hardware.

Example: Typing ^2o7 will enter one character into the text.

CTRL-P	('220)	is interpreted as a .BREAK.
.CR.	('215)	is interpreted as a newline (.NL.)
"	('242)	is interpreted as a character erase
?	('277)	is interpreted as line kill
\	('o34)	is interpreted as a logical tab (Editor)

Table C-1. ASCII Character Set (Non-Printing)

Octal Value	ASCII Char	Comments/Prime Usage	Control Char
200	NULL	Null character - filler	^@
201	SOH	Start of header (communications)	^A
202	STX	Start of text (communications)	^B
203	ETX	End of text communications	^C
204	EOT	End of transmission (communications)	^D
205	ENQ	End of I.D. (communications)	^E
206	ACK	Acknowledge affirmative (communications)	^F
207	BEL	Audible alarm (bell)	^G
210	BS	Back space one position (carriage control)	^H
211	HT	Physical horizontal tab	^I
212	LF	Line feed; ignored as terminal input	^J
213	VT	Physical vertical tab (carriage control)	^K
214	FF	Form feed (carriage control)	^L
215	CR	Carriage return (carriage control) (1)	^M
216	SO	RRS-red ribbon shift	^N
217	SI	BRS-black ribbon shift	^O
220	DLE	RCP-relative copy (2)	^P
221	DC1	RHT-relative horizontal tab (3)	^Q
222	DC2	HLF-half line feed forward (carriage control)	^R
223	DC3	RVT-relative vertical tab (4)	^S
224	DC4	HLR-half line feed reverse (carriage control)	^T
225	NAK	Negative acknowledgement (communications)	^U
226	SYN	Synchronocity (communications)	^V
227	ETB	End of transmission block (communications)	^W
230	CAN	Cancel	^X
231	EM	End of Medium	^Y
232	SUB	Substitute	^Z
233	ESC	Escape	^[
234	FS	File separator	^\
235	GS	Group separator	^]
236	RS	Record separator	^^
237	US	Unit separator	^_

Notes

(1) Generally, CR is interpreted as ".L." at the terminal. In Pascal, however, CR(or LF) always returns as a blank.

(2) ".BREAK." at terminal. Relative copy in file; next byte specifies number of bytes to copy from corresponding position of preceeding line.

(3) Next byte specifies number of spaces to insert.

(4) Next byte specifies number of lines to insert.

Conforms to ANSI X3.4-1968

The parity bit (°200) has been added for Prime-usage.

Non-printing characters (^c) can be entered at most terminals by typing the (control) key and the c character key simultaneously.

Table C-2. ASCII Character Set (Printing)

Octal Value	ASCII Character	OCTAL Value	ASCII Character	OCTAL Value	ASCII Character
240	.SP (1)	300	@	340	` (9)
241	!	301	A	341	a
242	" (2)	302	B	342	b
243	# (3)	303	C	343	c
244	$	304	D	344	d
245	%	305	E	345	e
246	&	306	F	346	f
247	' (4)	307	G	347	g
250	(	310	H	350	h
251	)	311	I	351	i
252	*	312	J	352	j
253	+	313	K	353	k
254	, (5)	314	L	354	l
255	-	315	M	355	m
256	.	316	N	356	n
257	/	317	O	357	o
260	0	320	P	360	p
261	1	321	Q	361	q
262	2	322	R	362	r
263	3	323	S	363	s
264	4	324	T	364	t
265	5	325	U	365	u
266	6	326	V	366	v
267	7	327	W	367	w
270	8	330	X	370	x
271	9	331	Y	371	y
272	:	332	Z	372	z
273	;	333	[	373	{
274	<	334	\	374	|
275	=	335	]	375	}
276	>	336	^ (7)	376	~ (10)
277	? (6)	337	_ (8)	377	DEL (11)

Notes

(1) Space forward one position

(2) Terminal usage - erase previous character

(3) in British use

(4) Apostrophe/single quote

(5) Comma

(6) Terminal usage - kill line

(7) 1963 standard ^ |; terminal use - logical escape

(8) 1963 standard <-; underscore "_"

(9) Grave

(10) 1963 standard ESC

(11) Rubout - ignored

Conforms to ANSI X3.4-1968
1963 variances are noted

*The parity bit ('200) has been added for Prime usage.