Pascal/VS

Reference Manual

May 12, 1980

Final Draft

IBM Internal Use Only

First Edition (May 1980)
This is the first edition of SH20-6163-0, a new publication that applies to the Pascal/VS Installed User Program, program number 5796-PNQ.

It is possible that this material may contain reference to, or information about IBM products (machines and programs) that are not available in your country. Such references or information must not be construed to mean that IBM intends to announce such products in your country.

Requests for copies of IBM publications should be made to the IBM branch office that serves you.

A form for reader's comments has been supplied at the back of this publication. If the form has been removed, comments may be addressed to Pascal/VS Development, IBM Corporation, M48/D25, P. O. BOX 50020, San Jose, California 95150. IBM may use or distribute any information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information that you supply.

(c) Copyright International Business Machines Corporation 1980

Preface

This document is the reference manual to the Pascal/VS programming language. The Pascal/VS Programmer's Guide, SH20-6162, is also available from IBM to help write programs in Pascal/VS.

It is assumed that you are already familiar with Pascal and programming in a high level programming language. There are many text books available on Pascal; the following list of books was taken from the Pascal User's Group Pascal News, December 1978 NUMBER 13 and September 1979 NUMBER 15. You may wish to check later editions of Pascal News and your library for more recent books.

- The Design of Well-Structured and Correct Programs by S. Alagic and M.A. Arbib, Springer-Verlag, New York, 1978, 292 pp.
- Microcomputer Problem Solving by K.L. Bowles, Springer-Verlag, New York, 1977, 563 pp.
- A Structured Programming Approach to Data, by D. Coleman, MacMillan Press Ltd, London, 1978, 222 pp.
- A Primer on Pascal by R.W. Conway, D. Gries and E.C. Zimmerman, Winthrop Publishers Inc., Cambridge Mass., 1976, 433 pp.
- PASCAL: An Introduction to Methodical Programming by W. Findlay and D. Watt, Computer Science Press, 1978, 306 pp.; UK Edition by Pitman International Text, 1978.
- Programming in PASCAL by Peter Grogono, Addison-Wesley, Reading Mass., 1978, 357pp.
- Pascal Users Manual and Report by K. Jensen and N. Wirth, Springer-Verlag, New York, 1978, 170 pp.
- Structured Programming and Problem-Solving with Pascal by R.B. Kieburtz, Prentice-Hall Inc., 1978, 365 pp.
- Programming via Pascal by J.S. Rohl and Barrett, Cambridge University Press.
- An Introduction to Programming and Problem-Solving with Pascal, by G.M. Schneider, S.W. Weingart and D.M. Perlman, Wiley & Sons Inc., New York, 394 pp.
- Introduction to Pascal, by C.A.G. Webster, Heyden, 1976, 129 pp.
- Introduction to Pascal, by J. Welsh and J. Elder, Prentice-Hall Inc., Englewood Cliffs, 220 pp.
- A Practical Introduction to Pascal by I.P. Wilson and A.M. Addyman, Springer-Verlag New York, 1978, 145pp; MacMillan, London, 1978.
- Systematic Programming: An Introduction by N. Wirth, Prentice-Hall Inc., Englewood Cliffs, 1973 169 pp.
- Algorithms + Data Structures = Programs by N. Wirth, Prentice-Hall Inc., Englewood Cliffs, 1976 366 pp.

This reference manual considers ISO/TC 97/SC 5 N565 as the Pascal Standard although N565 is a proposed standard and subject to further modification.

Structure of this Manual

This manual is divided into the following major topics

Chapter 1 is a summary of the language.

Chapter 2 is a description of the basic units (lexical) of Pascal/VS.

Chapters 3 through 9 are a top-down presentation of the language.

Chapter 10 describes the I/O procedures and functions.

Chapter 11 describes the predefined procedures and functions.

Chapter 12 describes the compiler directives.

Appendices provide supplemental information about Pascal/VS.

Pascal/VS Syntax Diagrams

The syntax of Pascal/VS will be described with the aid of syntax diagrams. These diagrams are essentially 'road maps'; by traversing the diagram in the direction of the arrows you can identify every possible legal Pascal/VS program.

Within the syntax diagram, the names of other diagrams are printed in lower case and surrounded by braces ('{}'). When you traverse the name of another diagram you can consider it a subroutine call (or more precisely a 'subdiagram call'). The names of reserved words are always in lower case. Special symbols (i.e. semicolons, commas, operators etc) appear as they appear in a Pascal/VS program.

The diagram traversal starts at the upper left and completes with the arrow on the right. Every horizontal line has an arrowhead to show the direction of the traversal on that line. The direction of traversal on the vertical lines can be deduced by looking at the horizontal lines to which it connects. Dashed lines (i.e. '----') indicate constructs which are unique to Pascal/VS and are not found in standard Pascal.

Identifiers may be classified according to how they are declared. For the sake of clarity, a reference in the syntax diagram for {id} is further specified with a one or two word description indicating how the identifier was declared. The form of the reference is '{id:description}'. For example {id:type} references an identifier declared as a type; {id:function} references an identifier declared as a function name.

Revision Codes

The convention used in this document is that all changes in the current version from the previous edition are flagged with a vertical bar in the left margin.

Extensions to Pascal are marked with a plus sign in the margin.

CONTENTS

1.0 INTRODUCTION TO PASCAL/VS

"The language Pascal was designed by Professor Niklaus Wirth to satisfy two principal aims:

(a) to make available a language suitable for teaching programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language.

(b) to define a language whose implementations could be both reliable and efficient on then available computers."

(Pascal Draft Proposal ISO/TC 97/SC 5 N565, February 19, 1980)

Pascal/VS is an extension to standard Pascal. The purpose of extending Pascal is to facilitate application programming requirements. Among the extensions are such features as separately compilable external routines, internal and external static data, and varying length character strings.

Pascal is of interest as a high level programming language for the following reasons:

- It provides constructs for defining data structures in a clear manner.
- It is suitable for applying structured programming techniques.
- The language is relatively machine-independent.
- Its syntax and semantics allow extensive error diagnostics during compilation.
- A program written in the language can have extensive execution time checks.
- Its semantics allow efficient object code to be generated.
- Its syntax allows relatively easy compilation.
- The language is relatively well known and is growing in popularity.

1.1 PASCAL LANGUAGE SUMMARY

This section of the manual is meant to be a capsule summary of Pascal/VS. It should serve as a brief outline to the language. The details are explained in the remainder of this document.

Modules	
program	self-contained and independently executable module
segment	a shell in which procedures and functions may be separately compiled

Declarations	
label	declares a label in a program, procedure or function
const	declares an identifier that becomes synonymous with a compile time computable value
type	declares an identifier which is a user-defined data type
var	declares a local variable
def	declares a variable which is defined in one module and may be referenced in other modules
ref	declares a variable which is defined in another module
static	declares a variable which is persists for the entire execution of the program
value	assigns a value to a def or static variable at compile time
procedure	a unit of a module which may be invoked
function	a unit of a module which may be invoked and returns a value

Data Types	
enumeration	a list of constants of a user-defined scalar data type
subrange	a continuous subset of a scalar type
array	a data structure composed of a list of homogeneous elements
record	a data structure composed of a list of heterogeneous elements
set	a collection of zero or more scalar values
file	a sequence of data to be read or written by a Pascal program
pointer	a reference to a variable created by the programmer

Predefined Data Type Identifiers	
INTEGER	whole numbers in the range -2147483648..2147483647
REAL	System/370 long floating point numbers
CHAR	an EBCDIC character
BOOLEAN	an enumerated scalar with values FALSE and TRUE
TEXT	a "file of char", used for readable input and output
ALFA	a "packed array[1..8] of char"
ALPHA	a "packed array[1..16] of char"
STRING	a "packed array[1..n] of char" where n varies up to compile time specified maximum value

Predefined Constant Identifiers	
FALSE	boolean constant
TRUE	boolean constant
MAXINT	value is equal to 2147483647 which is the largest INTEGER value
MININT	value is equal to -2147483648 which is the smallest INTEGER value
ALFALEN	value is equal to 8 which the number of characters in an ALFA
ALPHALEN	value is equal to 16 which the number of characters in an ALPHA

Parameter Passing Mechanisms	
value	parameter passing method whereby a copy of the actual parameter is assigned to the formal parameter
variable	parameter passing method whereby the formal parameter represents the variable which is the actual parameter; this method is also refered to as by reference
constant	parameter passing method whereby the formal parameter is treated as if it were a constant
procedural	the mechanism whereby a procedure may be passed to the called routine and executed from there
functional	the mechanism whereby a function may be passed to the called routine and executed from there

Executable Statements	
assert	a statement that permits you to specify a condition that should be true and if not causes a runtime error to be indicated
assignment	the statement that assigns a value to a variable
case	this statement causes any one of a list of statements to be executed based upon the value of an expression
compound	the 'begin/end' reserved words bracket a series of statements that cause the series to act as a single statement
continue	this statement resumes execution of the next iteration of the innermost loop. The termination condition is tested to determine if the loop should continue
empty	the statement that contains no executable code
for loop	a looping statement that modifies a control variable for each iteration of the loop
goto	the statement which changes the flow of your program
if	this statement causes one of two statements to be executed based on the evaluation of an expression
leave	this statement terminates the execution of the innermost loop. Execution resumes as if the loop termination condition were true
procedure call	this statement invokes a procedure. At the conclusion of the procedure, execution continues at the next statement
repeat loop	a loop with the termination test after each execution of the iterated statements
return	this statement terminates execution of the executing routine and returns control to the caller
while loop	a loop with the termination test before each execution of the iterated statement
with	a statement that permits complicated references to fields within a record to be simplified

<table>
<tr><th colspan="4">Multiplying Operators</th></tr>
<tr><th>operator</th><th>operation</th><th>operands</th><th>result</th></tr>
<tr><td>*</td><td>multiplication</td><td>INTEGER
REAL
one REAL, one INTEGER</td><td>INTEGER
REAL
REAL</td></tr>
<tr><td>/</td><td>real division</td><td>INTEGER
REAL
one REAL, one INTEGER</td><td>REAL
REAL
REAL</td></tr>
<tr><td>div</td><td>integer division</td><td>INTEGER</td><td>INTEGER</td></tr>
<tr><td>mod</td><td>modulo</td><td>INTEGER</td><td>INTEGER</td></tr>
<tr><td>& (and)</td><td>boolean and</td><td>BOOLEAN</td><td>BOOLEAN</td></tr>
<tr><td>& (and)</td><td>logical and</td><td>INTEGER</td><td>INTEGER</td></tr>
<tr><td>*</td><td>set intersection</td><td>set of t</td><td>set of t</td></tr>
<tr><td>||</td><td>string catenation</td><td>STRING</td><td>STRING</td></tr>
<tr><td><<</td><td>logical left shift</td><td>INTEGER</td><td>INTEGER</td></tr>
<tr><td>>></td><td>logical right shift</td><td>INTEGER</td><td>INTEGER</td></tr>
</table>

<table>
<tr><th colspan="4">Adding Operators</th></tr>
<tr><th>operator</th><th>operation</th><th>operands</th><th>result</th></tr>
<tr><td>+</td><td>addition</td><td>INTEGER
REAL
one REAL, one INTEGER</td><td>INTEGER
REAL
REAL</td></tr>
<tr><td>-</td><td>subtraction</td><td>INTEGER
REAL
one REAL, one INTEGER</td><td>INTEGER
REAL
REAL</td></tr>
<tr><td>-</td><td>set difference</td><td>set of t</td><td>set of t</td></tr>
<tr><td>| (or)</td><td>boolean or</td><td>BOOLEAN</td><td>BOOLEAN</td></tr>
<tr><td>| (or)</td><td>logical or</td><td>INTEGER</td><td>INTEGER</td></tr>
<tr><td>+</td><td>set union</td><td>set of t</td><td>set of t</td></tr>
<tr><td>&& (xor)</td><td>boolean xor</td><td>BOOLEAN</td><td>BOOLEAN</td></tr>
<tr><td>&& (xor)</td><td>logical xor</td><td>INTEGER</td><td>INTEGER</td></tr>
<tr><td>&& (xor)</td><td>'exclusive' union</td><td>set of t</td><td>set of t</td></tr>
</table>

<table>
<tr><th colspan="4">The Not Operator</th></tr>
<tr><th>operator</th><th>operation</th><th>operand</th><th>result</th></tr>
<tr><td>¬ (not)</td><td>boolean not</td><td>BOOLEAN</td><td>BOOLEAN</td></tr>
<tr><td>¬ (not)</td><td>logical one's complement</td><td>INTEGER</td><td>INTEGER</td></tr>
<tr><td>¬ (not)</td><td>set complement</td><td>set</td><td>set</td></tr>
</table>

Relational Operators			
operator	operation	operands	result
=	compare equal	any set, scalar type, pointer or string	BOOLEAN
<> (¬=)	compare not equal	any set, scalar type, pointer or string	BOOLEAN
<	compare less than	scalar type or string	BOOLEAN
<=	compare < or =	scalar type, string	BOOLEAN
<=	subset	set of t	BOOLEAN
>	compare greater	scalar type, string	BOOLEAN
>=	compare > or =	scalar type, string	BOOLEAN
>=	superset	set of t	BOOLEAN
in	set membership	t and set of t	BOOLEAN

Reserved Words			
and	end	not	+ segment
array	file	of	set
+ assert	for	or	+ space
begin	function	+ otherwise	+ static
case	goto	packed	then
const	if	procedure	to
+ continue	in	program	type
+ def	label	+ range	until
div	+ leave	record	+ value
do	mod	+ ref	var
downto	nil	repeat	while
else		+ return	with
			+ xor

note: those words marked by '+' are not reserved in standard Pascal

<table>
<tr><th colspan="2">Special Symbols</th></tr>
<tr><th>symbol</th><th>meaning</th></tr>
<tr><td>+</td><td>addition and set union operator</td></tr>
<tr><td>-</td><td>subtraction and set difference operator</td></tr>
<tr><td>*</td><td>multiplication and set intersection operator</td></tr>
<tr><td>/</td><td>division operator, REAL results only</td></tr>
<tr><td>¬</td><td>BOOLEAN not, one's complement on INTEGER or set complement</td></tr>
<tr><td>|</td><td>BOOLEAN or, logical or on INTEGER</td></tr>
<tr><td>&</td><td>BOOLEAN and, logical and on INTEGER</td></tr>
<tr><td>&&</td><td>BOOLEAN xor operator, logical xor on INTEGER and set exclusive union</td></tr>
<tr><td>=</td><td>equality operator</td></tr>
<tr><td><</td><td>less than operator</td></tr>
<tr><td><=</td><td>less than or equal operator</td></tr>
<tr><td>>=</td><td>greater than or equal operator</td></tr>
<tr><td>></td><td>greater than operator</td></tr>
<tr><td><> or ¬=</td><td>not equal operator</td></tr>
<tr><td>>></td><td>right logical shift on INTEGER</td></tr>
<tr><td><<</td><td>left logical shift on INTEGER</td></tr>
<tr><td>||</td><td>catenation operator</td></tr>
<tr><td>:=</td><td>assignment symbol</td></tr>
<tr><td>.</td><td>period to end a module</td></tr>
<tr><td>.</td><td>field separator in a record</td></tr>
<tr><td>,</td><td>comma, used as a list separator</td></tr>
<tr><td>:</td><td>colon, used to specify a definition</td></tr>
<tr><td>;</td><td>semicolon, used as a statement separator</td></tr>
<tr><td>..</td><td>subrange notation</td></tr>
<tr><td>'</td><td>quote, used to begin and end string constants</td></tr>
<tr><td>@ or -></td><td>pointer symbol</td></tr>
<tr><td>(</td><td>left parenthesis</td></tr>
<tr><td>)</td><td>right parenthesis</td></tr>
<tr><td>[or (.</td><td>left square bracket</td></tr>
<tr><td>] or .)</td><td>right square bracket</td></tr>
<tr><td>{ or (*</td><td>comment left brace (standard)</td></tr>
<tr><td>} or *)</td><td>comment right brace (standard)</td></tr>
<tr><td>/*</td><td>comment left brace (alternate form)</td></tr>
<tr><td>*/</td><td>comment right brace (alternate form)</td></tr>
</table>

Constants	
integer	whole numbers in the range -2147483648..2147483647
real	System/370 long (8 bytes) floating point numbers
string	a sequence of EBCDIC characters
nil	the value of a pointer which does not point to a variable
array constant	constant of an array type
record constant	constant of a record type
set constant	constant of a set type

2.0 THE BASE VOCABULARY

2.1 IDENTIFIERS

Syntax:

id:

```
                          ┌──────────────────────────────────────────────────────>
                          │
                          ├───>{digit}─────────>┐
 ──>{letter}──────┬───────┼───>{letter}────────>┼────>┐
                  │       └---->{underscore}-->┘     │
                  │                                   │
                  └<──────────────────────────────────┘
```

where:
{letter} is 'A', 'B', ..., 'Z','a','b', ... 'z' or '$'
{digit} is '0', '1', ..., '9'
underscore is '_'

Identifiers are names given to variables, data types, procedures, functions, named constants and modules.

<u>correct</u>:	<u>incorrect</u>:
I	5K
K9	NEW JERSEY
New_York	
AMOUNT$	

Valid and Invalid Identifiers

Pascal/VS permits identifiers of up to 16 characters in length. You may use longer names but Pascal/VS will ignore the portion of the name longer than 16 characters. You must assure identifiers are unique within the first 16 positions.

There is no distinction between lower and upper case letters within an identifier name. For example, the names 'ALPHA', 'alpha', and 'Alpha' are equivalent.

There is an implementation restrictions on the naming of external variables and external routines. You must make sure that identifiers used as external names are unique in the first 8 characters.

2.2 LEXICAL SCOPE OF IDENTIFIERS

The area of the module where a particular identifier can be referenced is called the <u>lexical scope</u> of the identifier (or simply scope).

In general, scopes are dependent on the structure of routine declarations. Since routines may be nested within other routines, a <u>lexical level</u> is associated with each routine. In addition, record definitions define a lexical scope for the fields of the record. Within a lexical level, each identifier can be defined only once. A program module is at level 0, routines defined within the module are at level 1; in general, a routine defined in level i would be at level (i+1). The following diagram illustrates a nesting structure.

```
program M (level 0)
  procedure A (level 1)
    procedure B (level 2)
      type
        R =
          record
          R1:...
          R2:...
          end;
      function C
            (level 3)
    procedure D (level 2)
  function X (level 1)
    procedure Y (level 2)
    procedure Z (level 2)
```

The scope of an identifier is the entire routine (or module) in which it was declared; this includes all routines defined within the routine. The following table references the preceding diagram.

identifiers declared in:	are accessible in:
Module M	M,A,B,C,D,X,Y,Z
procedure A	A,B,C,D
procedure B	B,C
type R	B,C
function C	C
procedure D	D
function X	X,Y,Z
procedure Y	Y
procedure Z	Z

If an identifier is declared in a routine which is nested in the scope of another identifier with the same name, then the new identifier will be the one recognized when its name appears in the routine. The first identifier becomes inaccessible in the routine. In other words, the identifier declared at the inner most level is the one accessible.

The scope of a field identifier defined within a record definition is limited to the record itself. The scope of a record may be accessed by either field referencing (section 7.2) or with the with-statement (section 9.15, page 90).

The Pascal/VS compiler effectively inserts a prelude of declarations at the beginning of every module it compiles. These declarations consist of the predefined types, constants, and routines. The scope of the prelude encompasses the entire module. You may re-declare any identifier that is predefined if you would like to use the name for another purpose.

2.3 RESERVED WORDS

Reserved Words			
and	end	not	+ segment
array	file	of	set
+ assert	for	or	+ space
begin	function	+ otherwise	+ static
case	goto	packed	then
const	if	procedure	to
+ continue	in	program	type
+ def	label	+ range	until
div	+ leave	record	+ value
do	mod	+ ref	var
downto	nil	repeat	while
else		+ return	with
			+ xor
note: those words marked by '+' are not reserved in standard Pascal			

Pascal/VS reserves the identifiers shown above for expressing the syntax of the language. These reserved words may never be declared by you. Reserved words must be separated from other reserved words and identifiers by a special symbol, a comment, or at least one blank.

A lower case letter is treated as equivalent to the corresponding upper case letter in a reserved word.

2.4 SPECIAL SYMBOLS

Special Symbols	
symbol	meaning
+	addition and set union operator
-	subtraction and set difference operator
*	multiplication and set intersection operator
/	division operator, REAL result only
¬	BOOLEAN not, one's complement on INTEGER or set complement
|	BOOLEAN or, logical or on INTEGER
&	BOOLEAN and, logical and on INTEGER
&&	BOOLEAN xor operator, logical xor on INTEGER and set exclusive union
=	equality operator
<	less than operator
<=	less than or equal operator
>=	greater than or equal operator
>	greater than operator
<> or ¬=	not equal operator
>>	right logical shift on INTEGER
<<	left logical shift on INTEGER
||	catenation operator
:=	assignment symbol
.	period to end a module
.	field separator in a record
,	comma, used as a list separator
:	colon, used to specify a definition
;	semicolon, used as a statement separator
..	subrange notation
'	quote, used to begin and end string constants
{ or ->	pointer symbol
(	left parenthesis
)	right parenthesis
[or (.	left square bracket
] or .)	right square bracket
{ or (*	comment left brace (standard)
} or *)	comment right brace (standard)
/*	comment left brace (alternate form)
*/	comment right brace (alternate form)

Special symbols used by Pascal/VS are listed above. Several special symbols may also be written as a reserved word. These symbols are shown in the following table.

Symbol	Reserved Word
¬	not
|	or
&	and
&&	xor

2.5 COMMENTS

Pascal/VS supports two forms of comments: '{ ... }' and '/*...*/'. The curved braces are the standard comment symbol in Pascal. The symbols '(*' and '*)' are considered by the compiler to identical to left and right braces. The form of comment using '/*' and '*/' is considered to be distinct from the form using braces.

When the compiler encounters the symbol '{', it will bypass all characters, including end-of-line, until the symbol '}' is encountered. Likewise, all characters following '/*' will be bypassed until the symbol '*/' is detected. As a result, either form may be used to enclose the other; for example /*...{...}...*/ is one comment. The two distinct forms of comments are useful to distingish different kinds of information to the program reader. For example, a '/*...*/' comment could be used to indicate a temporary piece of code, or perhaps debugging statements.

A comment may be placed anywhere in a module where a blank would be acceptable.

```
/*
if A = 10 then (* this statement is
                  for program
                  debugging        *)
  WRITE('A IS EQUAL TO TEN');
*/
```

Example of a nested Comment

2.6 CONSTANTS

Syntax:

unsigned-integer:

```
─────┬────┬──>{digit}──┬─────────────────────────────────┬──────────>
     │    └<───────────┘                                 │
     ├───> ' ───┬───>{binary digit}───┬───> 'B ───>──────┤
     │          └<────────────────────┘                  │
     └───> ' ───┬───>{hex-digit}───┬───> 'X ──────>──────┘
                └<─────────────────┘
```

real-number:

```
─────┬───> ' ───┬───>{hex-digit}───┬───> 'XR ─────────────────>┐
     │          └<─────────────────┘                           │
     └────┬──>{digit}──┬────────┬───> . ───┬──>{digit}──┬──┬───┼──────>
          │    └<──────┘        │          └<───────────┘  │   │
          ├<────────────────────┘                          │   │
          ├<───────────────────────────────────────────────┘   │
          │                                                    │
          └───> E ───┬──────────>┬────┬──>{digit}──────────>───┘
                     ├──> + ────>┤    └<──────────┘
                     └──> - ────>┘
```

unsigned-number:

```
─────┬───>{unsigned-integer}───>┬──────────────────────────────>
     └───>{real-number}────────>┘
```

string:

```
─────┬───> ' ─────┬───────────────────┬───> ' ────┬────────────>
     │            └<──{character}<────┘           │
     └───> ' ───┬───>{hex-digit}───┬───> 'XC ────>┘
                └<─────────────────┘
```

unsigned-constant:

```
─────┬───>{unsigned-number}──┬──────────────────────────────────>
     ├───>{string}─────────>─┤
     ├───>{id:constant}────>─┤
     └───> nil ────────────>─┘
```

constant:

```
─────┬───>{unsigned-constant}──────────────────────────┬────────>
     └────┬──> + ───┬──>{unsigned-number}─────>────────┘
          └──> - ──>┘
```

where:

{binary-digit} is '0' or '1'.
{digit} is '0' through '9';
{hex-digit} is '0' through '9' and 'A' through 'F';
{character} is any EBCDIC character.

Constants can be divided into several categories according to the predefined type to which they belong. An unsigned number will conform to either a REAL or an INTEGER. Strings will conform to the type STRING or packed array[1..n] of CHAR. In addition, if the string is one character in length, it will conform to the type CHAR.

If a single quote is to be used within a string, then the quote must be written twice. Lower case and upper case letters are distinct within string constants. String literals are not permitted to extend past the end of line of a source line. Longer strings can be formed by catenating shorter strings.

Nil is of a special type which will conform to any pointer type. It represents a unique pointer value which is not a valid address.

The constants TRUE and FALSE are predefined in the language and are of the standard type BOOLEAN.

Integer hexadecimal constants are enclosed in quotes and suffixed with an 'X' or 'x'. Integer binary constants are enclosed in quotes and suffixed with a 'B' or 'b'. Such constants may be used in any context where an integer constant is appropriate. If you do not specify 8 hexadecimal digits (i.e. 4 bytes), Pascal/VS assumes that the digits not supplied are zeros on the left. For example, 'F'x is the value 15.

Floating point hexadecimal constants are enclosed in quotes and suffixed with an 'XR' or 'xr'. Such constants may be used in any context where a real constant is appropriate. If you do not specify 16 hexadecimal digits (i.e. 8 bytes), Pascal/VS assumes that the digits not supplied are zeros on the right. For example, '4110'xr is the same as '4110000000000000'xr.

String hexadecimal constants are enclosed in quotes and suffixed with an 'XC' or 'xc'. Such constants may be used in any context where a string constant is appropriate. There must be an even number of digits within a hexadecimal string constant; that is, you must specify each character fully that is to be in the string.

The symbol 'E' or 'e' when used in a real-number expresses 'ten to the power of'.

Pascal/VS permits <u>constant expressions</u> in places where the Pascal standard only permits constants. Constant expressions are evaluated and replaced by a single result at compile time. See section 8.2 on page 66 for a description of constant expressions.

<u>constant</u> <u>matches</u>	<u>standard type</u>
`0`	INTEGER
`-500`	INTEGER
`1.0`	REAL
`314159E-5`	REAL
`0E0`	REAL
`1.0E10`	REAL
`TRUE`	BOOLEAN
`'FF'X`	INTEGER
`'A'`	CHAR
`'ABC'`	STRING
`'C1C2C2'xc`	STRING
`'4E800000FFFFFFFF'xr`	REAL
`'abc'`	STRING
`''`	STRING
`''''`	CHAR
`' '`	CHAR
`'  '`	STRING
`'Thats''s all '`	STRING

Examples of Constants

2.7 STRUCTURED CONSTANTS

```
Syntax:

structured-constant:

---┬--->{record-structure}---┬------------------------------------------>
   └--->{array-structure}--->┘

record-structure:

--->{id:type}---> ( ---┬---┬--->{constant-expr}---┬---┬---> ) ----------->
                       │   └--------------------->┘   │
                       └<------------ , <-------------┘

array-structure:

--->{id:type}---> ( --->┐
   ┌<-------------------┘
   └--┬---┬--->{constant-expr}--┬---> : {repetition}--┬---┬---┬---> ) ---------->
      │   │                     └-------------------->┘   │   │
      │   └---------------------------------------------->┘   │
      └<------------------ , <--------------------------------┘

repetition:

--->{constant-expr}---------------------------------------------------->

note: must evaluate to positive integer.
```

Constant structures are constants which are of a structured type. The type of the constant is determined by the type identifier which is used in its definition. These constants may be used in constant declarations, value declarations or in the executable statements.

There are two kinds of constant structures: one is used for arrays and the second is used to specify records.

Array constants are specified by a list of constant expressions where each expression defines one element of the array. See section 8.2 on page 66 for a description of constant exepressions. You may omit an element of the array within the list in which case the value of that element is not defined. Elements may be omitted at the end of the array in which case the value of those elements are also not defined. You may follow the constant expression with a colon and a repetition expression; this is used to specify that the first constant expression is to be repeated.

The second kind of constant structure is used to specify records. Record constants are specified by a list of constant expressions where each expression defines one field of the record. You specify one constant expression for each field of the fixed part of the record. You may omit a field of the record within the list in which case the value of that field is not defined. Restriction: You may not specify a value for any fields in a variant part; note that the tagfield is considered part of the fixed part.

```
type
  COMPLEX = record
              RE,IM: REAL
            end;
  VECTOR  = array[1..7] of INTEGER;
  CARRAY  = array[0..9] of COMPLEX;

const
  (* Structured Constants      *)
  THREEFOUR = COMPLEX(3.0,4.0);
  VECTOR_1  = VECTOR(7,0:5,1);
  VECTOR_2  = VECTOR(2,3,,4);
  VECTOR_3  = CARRAY(
                COMPLEX(1.0,0.0),
                COMPLEX(1.0,1.0):8,
                COMPLEX(0.0,1.0));
```

Examples of Structured Constants

3.0 STRUCTURE OF A MODULE

Syntax:

module:

```
──────┬──>{program-module}─────┬──────────────────────────────────────────>
      └--->{segment-module}--->┘
```

program-module:

```
───> program ───>{id}─────┬───> ( ─────┬──>{id}───┬────> ) ─────┬───>┐
                          │            └<──  ,  <──┘             │    │
                          └--------------------------------------┘    │
      ┌<─────────────────────────────────────── ; <───────────────────┘
      │
      ├<──────────────────────────┐
      ├───>{declaration}─────>────┘
      │
      └───>{compound-statement}───> . ──────────────────────────────────────>
```

declaration:

```
──────┬───>{label-dcl}──────────>┐
      ├───>{constant-dcl}───────>┤
      ├───>{type-dcl}───────────>┤
      ├───>{var-dcl}────────────>┤
      ├--->{def-dcl}----------->-┤
      ├--->{static-dcl}-------->-┤
      ├--->{value-dcl}--------->-┤
      └───>{routine-dcl}────────>┴──────────────────────────────────────────>
```

segment-module:

```
---> segment --->{id}---> ; --->┐
      ┌<------------------------┘
      ├--->{constant-dcl}----->┐
      ├--->{type-dcl}--------->┤
      ├--->{def-dcl}---------->┤
      ├--->{static-dcl}------->┤
      ├--->{value-dcl}-------->┤
      ├--->{routine-dcl}------>┘
      └---> . ---------------------------------------------------------->
```

A module is an independently compilable unit of code. There are two types of modules in Pascal/VS: the program module and the segment module.

The program is the module which gains initial control when the compiled program is invoked from the system loader. It is effectively a procedure that the loader invokes. The body of a program module is identical to the body of a procedure.

A segment module may be compiled as a unit independent of the program module. It consists of routines that are to be linked into the final program prior to execution. Data is passed to routines through parameters and external variables. Segments are useful in breaking up large Pascal/VS programs into smaller units.

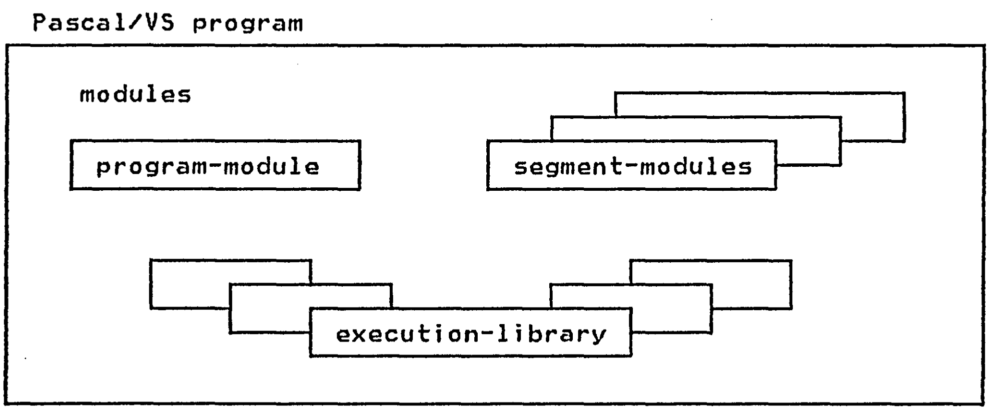
The identifier following the reserved word "program" must be a unique external name. The identifier following the reserved word "segment" may be the same as one of the ENTRY routines in the segment or may be a unique external name. Thus, an entry function called SIN could be in a segment called SIN. An external name is an identifier for a program, segment, def or ref variable, ENTRY routine, or EXTERNAL routine.

The optional identifier list following the program identifier is not used by Pascal/VS. The identifiers will be ignored.

A program is formed by linking a program module with segment modules (if any) and with the Pascal/VS execution library and libraries that you may supply.

Pascal/VS allows declarations to be given in any order. This is an extension to Pascal and is provided primarily to permit source that is INCLUDEd during compilation to be independent of any ordering already established in the module. The standard ordering for declarations is shown in the diagram for declarations. (For a description of the INCLUDE facility see section 12.1 on page 134)

Every identifier must be predefined or declared by you before it is used. There is one exception to this rule: a definition of a pointer may refer to an identifier before it is declared. The identifier must be declared later or a compile-time' diagnostic will be produced.




```
program EXAMPLE;
var
  I : INTEGER;
begin
  for I:=0 to 1000 do
    if I mod 7 = 0 then
      WRITELN( I:5,
         ' IS DIVISIBLE BY SEVEN')
end.
```

Example of a Program Module

```
segment EXAMSEG;
function COSINE
          (X:REAL ):REAL; ENTRY;
  var S: REAL;
  begin
    S := SIN(X);
    COSINE := SQRT(1.0 - S*S)
  end; .
```

Example of a Segment Module

4.0 PASCAL/VS DECLARATIONS

Pascal/VS provides you with 10 types of declarations:

- label
- const
- type
- var
- def +
- ref +
- static +
- value +
- procedure
- function

4.1 THE LABEL DECLARATION

Syntax:

label-dcl:

```
——> label ——┬——>{label}——┬——> ; ——————————————————————————————>
            └<—————— , <——┘
```

label:

```
——————┬——>{unsigned-integer}——┬——————————————————————————————>
+     └--->{id}--------------->┘
```

Note: the values of the unsigned integer must be in the subrange 0..9999.

A label declaration is used to declare labels which will appear in the routine and will be referenced by a goto statement within the routine. All labels defined within a routine must be declared in a label declaration within the routine.

A label may be either an unsigned integer or an identifier. If the value is an unsigned integer it must be in the range 0 to 9999.

```
label
  10,
  Label_A,
  1,
  2,
  Error_exit;
```

A Label Declaration

4.2 THE CONST DECLARATION

Syntax:

constant-dcl:

```
+  ──> const ──┬──>{id}──> = ──>{constant-expr}──> ; ──┬──────────────>
               └<──────────────────────────────────────┘
```

A constant declaration allows you to
+ assign identifiers that are to be used
+ as synonyms for constant expressions.
The type of a constant identifier is determined by the type of the expression in the declaration.

```
const
  BLANK     = ' ';
  BLANKS    = '   ';
  FIFTY     = 50;
  A         = FIFTY;
  B         = FIFTY * 10/(3+2);
  C_SQUARED = A*A + B*B;
  ORD_OF_A  = ORD('A');
  PI        = 3.14159265358;
  MASK      = '8000'X | '0400'X;
  ALFALEN   = 8;
  ALPHALEN  = 16;
  LETTERS   = [ 'A'..'Z','a'..'z' ]
  MAXREAL   = '4FFFFFFFFFFFFFFF'xr;
```

Constant Declarations

4.3 THE TYPE DECLARATION

Syntax:

type-dcl:

```
──> type ──┬──>{id}──> = ──>{type}──> ; ──┬─────────────────────>
           └<──────────────────────────────┘
```

A type declaration allows you to define a data type and associate a name to that type. Once declared, such a name may be used in the same way as a predefined type name.

```
type

   (* all of the following types    *)
   (* are predefined in Pascal/VS   *)

   INTEGER = MININT..MAXINT;
   BOOLEAN = (FALSE,TRUE);
   ALFA    = packed array[1..ALFALEN]
                of CHAR;
   ALPHA   = packed array[1..ALPHALEN]
                of CHAR;
   TEXT    = file of CHAR;
```

Type Declarations

4.4 THE VAR DECLARATION

Syntax:

var-dcl:

```
——> var ——┬——┬——>{id}——┬——> : ——>{type}——> ; ——┬————————————>
          │  └<—— , ———┘                        │
          └<————————————————————————————————————┘
```

The var declaration is used to declare automatic variables. Automatic variables are allocated when the routine is invoked, and are de-allocated when the corresponding return is made. If the routine is invoked a second time, before an initial invocation completes (a recursive call), the local automatic variables will be allocated again in a stack-like manner. The variables allocated for the first invocation become inaccessible until the recursive call completes.

Commas are used in the declaration to separate two or more identifiers that are being declared of the same type. This is a shorthand notation for two separate declarations.

You may have automatic variables declared in the outermost nesting (level 0) of a program module. However this is not the case for segment modules. This is because a segment is used as a shell in which procedures are compiled and has no activation (call) of its own. You may always declare static, def and ref variables in the outermost level of a segment.

```
var
 I      : INTEGER;
 SYSIN  : TEXT;
 X,
 Y,
 Z      : REAL;
 CARD   :
    record
      RANK : 1..13;
      SUIT : (SPADE,HEART,DIAMOND,CLUB)
    end;
```

Example of a Var Declaration

4.5 THE STATIC DECLARATION

Syntax:

static-dcl:

```
---> static ---T---T---->{id}---T---> : --->{type}---> ; ---T----------------->
               |   L<--- , -----J                           |
               |                                            |
               L<-------------------------------------------J
```

The static declaration is used to declare static variables. The variables declared in this way are allocated prior to program execution and exist for the life of the program's execution.

Static variables can be referenced according to the lexical scoping rules. Two static variables in different scopes are different variables even though they have the same name.

Data in static variables that are local to a routine will be preserved over separate invocations of the routine. Such a routine called recursively will access the same instance of each static variable.

Static variables may be initialized at compile-time by the use of a value declaration.

```
static
   SYSPRINT : TEXT;
   X,Y:       REAL;
```

Example of a Static Declaration

4.6 THE DEF/REF DECLARATION

Syntax:

def-dcl:

```
---T---> def ---T-----T-----T---->{id}---T---> : --->{type}---> ; ---T---------->
   L---> ref --->J    |     L<--- , ----J                             |
                      |                                               |
                      L<----------------------------------------------J
```

The def/ref declarations are used to declare external variables. External variables are allocated prior to execution and can be accessed from more than one module. All identifiers that are to be used as external names must be unique in the first eight characters.

If an external variable with a particular name is declared in several modules, a single common storage location will be associated with each such variable. An external variable must be declared with identical types in each module; the programmer is responsible for assuring that the types are the same.

The def declaration specifies that the program loader is responsible for generating the common storage for the variable. The ref declaration specifies that storage for the variable is defined in another module (or in the runtime environment). Ref declared variables will remain unresolved until the encompassing module is compiled and linked with a module in which the variable is declared as a def variable or defined in a non-Pascal CSECT or in an assembly language COM. The expected use of ref variables is to access external data declared in non-Pascal/VS programs such as those written in assembly language.

A def or ref variable may be declared local to a routine; the same scope rules apply as for any other declared identifier. However, if the name of the variable is declared in another scope (even in another module) as a def or ref variable, both occurrences of the variable will reference the same storage.

In the following example, the variable X in procedures A, B, and C references the same storage; however, the variables X declared in segment P and procedure D each refer to storage that is separate from the external variable X.

Def variables may be initialized at compile-time by the use of a value declaration.

```
segment M;
procedure A;
  def X: REAL;   (* same as X in B   *)
  begin
   ...
  end;

procedure B;
  def X: REAL;   (* same as X in A   *)
  begin
   ...
  end;.


 segment P;
 static X: REAL;(* local to P       *)
 procedure C;

   ref X: REAL; (* same as X in A,B*)
   begin
    ...
   end;

 procedure D;
   var X: REAL; (* local to D       *)
   begin
    ...
   end;.
```

Examples of Def and Ref Declarations

4.7 THE VALUE DECLARATION

```
Syntax:

value-dcl:

---> value ---┬---{value-assignment}---> ; ---┬----------------------------->
              └<-------------------------------┘

value-assignment:

--->{variable}---> := ---┬--->{constant-expression}----┬-------------------->
                         └--->{structured-constant}--->┘

note: If the variable contains subscripts, the subscripts are limited
      to constant expressions.
```

The value declaration is used to specify an initial value for static and def variables. The declaration is composed of a list of value-assignment statements separated by semicolons. The assignment statements in a value declaration are of the same form as the assignment statements in the body of a routine except that all subscripts and expressions must be able to be evaluated at compile time.

If a def variable is initialized with a value declaration in one module, you may not use a value declaration on that variable in another module. The compiler will not check this violation, however a diagnostic will be generated when you combine the modules into a single load module by the system loader.

```
type
  COMPLEX = record
               RE,IM: REAL
            end;
  VECTOR  = array[1..7] of INTEGER;

static
  C: COMPLEX;
  V: VECTOR;
  V1: VECTOR;

def
  I : INTEGER;
  Q : array[1..10] of COMPLEX;

(* the following assignments will *)
(* take place at compile time     *)
value
  C        := COMPLEX(3.0,4.0);
  V        := VECTOR(1,0:5,7);
  V1       := VECTOR(,,,4);
  V[2]     := 2;
  V[3]     := 3*4-1;
  I        := 0;
  Q[1].RE  := 3.1415926 / 2;
  Q[1].IM  := 1.414;
```

Example of a Value Declaration

5.0 TYPES

```
Syntax:

type:

───┬──>{id:type}──────────────────────────────────────────>┐
   ├──>{enumerated-scalar-type}───────────────────────────>┤
   ├──>{subrange-scalar-type}─────────────────────────────>┤
   ├──>{array-type}───────────────────────────────────────>┤
   ├──>{record-type}──────────────────────────────────────>┤
   ├──>{set-type}─────────────────────────────────────────>┤
   ├──>{file-type}────────────────────────────────────────>┤
   └──>{pointer-type}─────────────────────────────────────>┴──────>
```

A data type determines the kind of values that a variable of that type can assume. Pascal/VS allows you to define new data types with the type declaration. The data type mechanism is a very important part of Pascal/VS. With it you can describe your data with great clarity.

There are several mechanisms that can be used to define a type; each mechanism allows the new data type to have certain properties. All data types can be classified as either scalar, pointer, or structured.

You define the data type of a variable when the variable is declared. A previous type declaration allows an identifier to be associated with that type. Such an identifier can be used wherever a type definition is needed: in a variable declaration (var, static, def, or ref), as a parameter, in a procedure or function, in a field declaration within a record definition, or in another type declaration.

+ ## 5.1 A NOTE ABOUT STRINGS

+ Standard Pascal defines a string as
+ 'packed array[1..n] of CHAR' where n is
+ fixed for every string at compile time.
+ Pascal/VS extends the notation of a
+ string to a allow n to vary during
+ execution from 0 up to a compile time
+ specified maximum value. Variables can
+ be declared as a string type by using
+ the predefined data type STRING.
+ Throughout this manual a reference to a
+ string is assumed to refer to an object
+ of the predefined type STRING as
+ opposed to simply a 'packed array[1..n]
+ of CHAR'.

5.2 TYPE COMPATIBILITY

Pascal/VS supports _strong typing_ of data. The strong typing permits Pascal/VS to check the validity of many operations at compile time; this helps to produce reliable programs at execution time. Strong typing puts strict rules on what data types are considered to be the same. These rules, called _type compatibility_, requires you to carefully declare data.

5.2.1 IMPLICIT TYPE CONVERSION

In general, Pascal/VS does not perform implicit type conversions on data. The only implicit conversions that Pascal/VS permits are:

1. An INTEGER will be converted to a REAL when one operand of a binary operation is an INTEGER and the other is a REAL.
2. An INTEGER will be converted to a REAL when assigning an INTEGER to a REAL variable.
3. An INTEGER will be converted to a REAL if it is used in a floating point divide operation ('/').

4. An INTEGER will be converted to a REAL if it is passed by value or passed by const to a parameter requiring a REAL value.

5. A string will be converted to a 'packed array[1..n] of CHAR' on assignment. The string will be padded with blanks on the right if it is shorter than the array to which it is being assigned. Truncation will raise a runtime error if checking is enabled.

6. A string being passed by value or passed by const to a formal parameter that requires a 'packed array[1..n] of CHAR' will be converted. The string will be padded with blanks on the right if it is shorter than the array to which it is being passed. Truncation will raise a runtime error if checking is enabled.

5.2.2 SAME TYPES

Two variables are said to be of the *same type* if the declaration of the variables:

- refer to the same type identifier;
- or, refer to different type identifiers which have been defined as equivalent by a type definition of the form:

  type T1 = T2

5.2.3 COMPATIBLE TYPES

Operations can be performed between two values that are of *compatible types*. Two types are said to be compatible if:

- the types are the same;
- one type is a subrange of the other or they are both subranges of the same type;
- both types are strings;
- one value is a string literal and the other is a 'packed array[1..n] of CHAR';
- one value is a string literal of one character and the other is a CHAR;
- they are set types with compatible base types;
- or, they are both 'packed array[1..n] of CHAR' with the same number of elements.

Furthermore, any object which is of a set type is compatible with the empty set. And, any object which is a pointer type is compatible with the value nil.

5.2.4 ASSIGNMENT COMPATIBLE TYPES

A value may be assigned to a variable if the types are *assignment compatible*. An expression E is said to be assignment compatible with variable V if:

- the types are same type and neither is a file type;
- V is of type REAL and E is compatible with type INTEGER;
- V is a compatible subrange of E and the value to be assigned is within the allowable subrange of V;
- V and E have compatible set types and all members of E are permissible members of V; or,
- V is a 'packed array[1..n] of CHAR' and E is a string.

```
type

   X       = array[ 1..10 ] of
                INTEGER;
   DAYS    = (MON, TUES, WED, THURS,
              FRI, SAT,  SUN);
   WEEKDAY = MON .. FRI;

var

      A : array[ 1..10 ] of
            INTEGER;
   B : array[ 1..10 ] of
            INTEGER;
   C,
   D : array[ 1..10 ] of
            CHAR;
   E : X;
   F : X;
   W1: DAYS;
   W2: WEEKDAY

   is   compatible
          with
  ----- -------------
   A      A
   B      B
   C      C, D
   D      D, C
   E      E, F
   F      F, E
   W1     W1, W2
   W2     W2, W1
```

Examples of Compatibility

5.3 THE ENUMERATED SCALAR

Syntax:

enumerated-scalar-type:

```
──> ( ──┬──>{id}──┬──> ) ──────────────────────────────────────>
        └<── , <──┘
```

An enumerated scalar is formed by listing each value that is permitted for a variable of this type. Each value is an identifier which is treated as a self-defining constant. This allows a meaningful name to be associated with each value of a variable of the type.

```
type
   DAYS      = (MON,  TUES, WED, THURS,
                FRI,  SAT,  SUN);

   MONTHS    = (JAN,  FEB,  MAR, APR,
                MAY,  JUN,  JUL, AUG,
                SEP,  OCT,  NOV, DEC);

var
   SHAPE     : (TRIANGLE, RECTANGLE,
                SQUARE,   CIRCLE);

   REC       : record
                SUIT: (SPADE, HEART,
                       DIAMOND, CLUB);
                WEEK: DAYS
               end;

   MONTH     : MONTHS;
```

Enumerated Scalars

An enumerated scalar type definition declares the identifiers in the enumeration list as constants of the scalar type being defined. The lexical scope of the newly defined constants is the same as that of any other identifier declared explicitly at the same lexical level.

These constants are ordered such that the first value is less than the second, the second less than the third and so forth. In the first example, MON < TUES < WED < ... < SUN. There is no value less than the first or greater than the last.

The following predefined functions operate on expressions of a scalar type (see the indicated section for more details):

	Function	Section	Page
	ORD	11.4.1	113
+	MAX	11.5.2	117
+	MIN	11.5.1	117
	PRED	11.5.3	118
	SUCC	11.5.4	118
+	LOWEST	11.3.3	111
+	HIGHEST	11.3.4	112

Notes:

1. Two enumerated scalar type definitions must not have any elements of the same name in the same lexical scope.

2. The standard type BOOLEAN is defined as (FALSE, TRUE).

5.4 THE SUBRANGE SCALAR

Syntax:

subrange-scalar-type:

```
──┬───> packed ───>┐
  ├<───────────────┘
  │
  └────┬───>{constant}───> .. ───>{constant-expr}──────────────────┬─────>
       │                                                           │
       └───> range ───>{constant-expr}───> .. ───>{constant-expr}───>┘
```

The subrange type is a subset of consecutive values of a previously defined scalar type. Any operation which is permissible on a scalar type is also permissible on any subrange of it.

A subrange is defined by specifying the minimum and maximum values that will be permitted for data declared with that type. For subranges that are packed, Pascal/VS will assign the smallest number of bytes required to represent a value of that type.

If the reserved word range is used in the subrange definition, then both the minimum and maximum values may be any expression that can be computed at compile time. If the range prefix is not employed then the minimum value of the range must be a simple constant.

The following predefined functions operate on expressions of a scalar type (see the indicated section for more details):

Function	Section	Page
ORD	11.4.1	113
MAX	11.5.2	117
MIN	11.5.1	117
PRED	11.5.3	118
SUCC	11.5.4	118
LOWEST	11.3.3	111
HIGHEST	11.3.4	112

Notes:

1. A subrange of the standard type REAL is not permitted.
2. The number of values in a subrange of type CHAR is determined by the collating sequence of the EBCDIC character set.
3. The lower bound of a subrange definition that is not prefixed with 'range' must be a simple constant instead of a generalized constant expression.

```
const
  SIZE          = 1000;

type
  DAYS          = (SU,  MO,  TU,  WE,
                   TH,  FR,  SA);
  MONTHS        = (JAN, FEB, MAR, APR,
                   MAY, JUN, JUL, AUG,
                   SEP, OCT, NOV, DEC);
  UPPER_CASE    = 'A' .. 'Z';
  ONE_HUNDRED   =  0 .. 99;
  CODES         = range
                    CHR(0)..CHR(255);
  INDEX         = packed 1 .. SIZE+1;

var
  WORK_DAY      : MO .. FR;
  SUMMER        : JUN .. AUG;
  SMALLINT      : packed 0..255;
  YEAR          : 1900 .. 2000;
```

Subrange Scalars

5.5 PREDEFINED SCALAR TYPES

5.5.1 THE TYPE INTEGER

The following table describes the operations and predefined functions that apply to values which are the standard type INTEGER.

INTEGER		
operation	form	description
+	unary	returns the unchanged result of the operand
+	binary	forms the sum of the operands
-	unary	negates the operand
-	binary	forms the difference of the operands
*	binary	forms the product of the operands
/	binary	converts the operands to REAL and produces the REAL quotient
div	binary	forms the integer quotient of the operands
mod	binary	forms the integer modulus of the operands
=	binary	compares for equality
<> or ¬=	binary	compares for inequality
<	binary	compares for less than
<=	binary	compares for less than or equal to
>=	binary	compares for greater than or equal to
>	binary	compares for greater than
¬	unary	returns one's complement on the operand
|	binary	returns 'logical or' on the operands
&	binary	returns 'logical and' on the operands
&&	binary	returns 'logical xor' on the operands
<<	binary	returns the left operand value shifted left by the right operand value
>>	binary	returns the left operand value shifted right by the right operand value
CHR(x)	function	returns a CHAR whose EBCDIC representation is x
PRED(x)	function	returns x-1
SUCC(x)	function	returns x+1
ODD(x)	function	returns TRUE if x is odd and FALSE otherwise
ABS(x)	function	returns the absolute value of x
SQR(x)	function	returns the square of x
FLOAT(x)	function	returns a REAL whose value is x
MIN( )	function	returns the minimum value of two or more operands
MAX( )	function	returns the maximum value of two or more operands
LOWEST(x)	function	returns MININT or the minimum value of the range if x is a subrange of INTEGER
HIGHEST(x)	function	returns MAXINT or the maximum value of the range if x is a subrange of INTEGER
SIZEOF(x)	function	returns the number of bytes required for a value of the type of x, which is always 1, 2 or 4

The type INTEGER is provided as a predefined type in Pascal/VS. This type represents the subset of whole numbers as defined below:

```
type
  INTEGER = MININT..MAXINT;
```

where MININT is a predefined INTEGER constant whose value is -2147483648 and MAXINT is a predefined INTEGER constant whose value is 2147483647. That is, the predefined type INTEGER represents 32 bit values in 2's complement notation.

Type definitions representing integer subranges may be prefixed with the reserved word "packed". For variables declared with such a type, Pascal/VS will assign the smallest number of bytes required to represent a value of that type. The following table defines the number of bytes required for different ranges of integers. For ranges other than those listed, use the first range that encloses the desired range.

Given a type definition T as:

```
type
  T = packed i..j;
```

Range of i .. j	Size in bytes	Alignment
0..255	1	BYTE
-128..127	1	BYTE
-32768..32767	2	HALFWORD
0..65535	2	HALFWORD
otherwise	4	FULLWORD

Notes:

1. The operations of div and mod are defined as:

   ```
   A div B = TRUNC(A/B), B<>0

   A mod B = A-B*(A div B), A>=0,B>0
   A mod B = B-abs(A) mod B, A<0,B>0
   ```

   B=0 when doing a div operation or B<=0 when doing a mod operation is defined as an error and will cause a runtime error message to be produced.

2. The following operators perform logical operations:

   ```
   <<  shift left logical
   >>  shift right logical
   ¬   1's complement
   |   logical inclusive or
   &   logical and
   &&  logical exclusive or
   ```

   The operands are treated as unsigned strings of binary digits. See section 8.4 for more details on logical expressions.

5.5.2 THE TYPE CHAR

The following table describes the operations and predefined functions that apply to the standard type CHAR.

CHAR		
operation	form	description
=	binary	compares for equality
<> or ¬=	binary	compares for inequality
<	binary	compares for left less than right
<=	binary	compares for left less than or equal to right
>=	binary	compares for left greater than or equal to right
>	binary	compares for left greater than right
ORD(x)	function	converts operand to an INTEGER based on ordering sequence of underlying character set.
PRED(x)	function	returns the preceding character in collating sequence
SUCC(x)	function	returns the succeeding character in collating sequence
STR(x)	function	converts the operand to a STRING
MIN( )	function	returns the minimum value of two or more operands
MAX( )	function	returns the maximum value of two or more operands
LOWEST(x)	function	returns the minimum value of the range of the character x
HIGHEST(x)	function	returns the maximum value of the range of the character x
SIZEOF(x)	function	returns the number of bytes required for a value of the type of a CHAR, which is always 1

The type CHAR is defined as a value from the EBCDIC character set. Variables of this type occupy one byte of memory and will be aligned on a byte boundary.

5.5.3 THE TYPE BOOLEAN

The following table describes the operations and predefined functions that apply to the standard type BOOLEAN.

	BOOLEAN		
	operation	form	description
	¬	unary	returns TRUE if the operand is FALSE, otherwise it returns FALSE
	&	binary	returns TRUE if both operands are TRUE
	|	binary	returns TRUE if either operand is TRUE
+	&&	binary	returns TRUE if either, but not both operands are TRU
	=	binary	compares for equality
	<> or ¬=	binary	compares for inequality
	<	binary	compares for left less than right
	<=	binary	compares for left less than or equal to right
	>=	binary	compares for left greater than or equal to right
	>	binary	compares for left greater than right
	ORD(x)	function	returns 0 if x is FALSE and 1 if x is TRUE
+	MIN( )	function	returns TRUE if all operands are TRUE
+	MAX( )	function	returns FALSE if all operands are FALSE
+	LOWEST(x)	function	returns the FALSE by definition
+	HIGHEST(x)	function	returns the TRUE by definition
+	SIZEOF(x)	function	returns the number of bytes required for a value
+			of the type of a BOOLEAN, which is always 1

Binary Operations on BOOLEAN					
	FALSE FALSE	FALSE TRUE	TRUE FALSE	TRUE TRUE	Name
=	TRUE	FALSE	FALSE	TRUE	Equivalence
<>	FALSE	TRUE	TRUE	FALSE	Exclusive Or
<	FALSE	TRUE	FALSE	FALSE	
<=	TRUE	TRUE	FALSE	TRUE	Implication
>=	TRUE	FALSE	TRUE	TRUE	
>	FALSE	FALSE	TRUE	FALSE	
&	FALSE	FALSE	FALSE	TRUE	And
|	FALSE	TRUE	TRUE	TRUE	Inclusive Or
&&	FALSE	TRUE	TRUE	FALSE	Exclusive Or

This type is predefined as:

```
type
   BOOLEAN = (FALSE,TRUE);
```

The type BOOLEAN is defined as a scalar whose values are FALSE and TRUE. Variables of this type will occupy one byte of memory and will aligned on a byte boundary. The relational operators form valid boolean functions as shown in the table of binary operations.

Pascal/VS will optimize the evaluation of BOOLEAN expressions involving '&' (and) and '|' (or) such that the right operand expression will not be evaluated if the result of the operation can be determined by evaluating the left operand. For more details see section 8.3 on page 67.

5.5.4 THE TYPE REAL

The following table describes the operations and predefined functions that apply to the standard type REAL.

REAL		
operation	form	description
+	unary	returns the value of the operand
+	binary	forms the sum of the operands
-	unary	negates the operand
-	binary	forms the difference of the operands
*	binary	forms the product of the operands
/	binary	forms the REAL quotient of the operands
=	binary	compares for equality
<> or ¬=	binary	compares for inequality
<	binary	compares for left less than right
<=	binary	compares for left less than or equal to right
>=	binary	compares for left greater than or equal to right
>	binary	compares for left greater than right
TRUNC(x)	function	returns the operand value truncated to an INTEGER
ROUND(x)	function	returns the operand value rounded to an INTEGER
ABS(x)	function	returns the absolute value of the operand
SIN(x)	function	returns the trigonometric sine of x (radians)
COS(x)	function	returns the trigonometric cosine of x (radians)
ARCTAN(x)	function	returns (radians) the arc tangent of x
LN(x)	function	returns the natural logarithm of x
EXP(x)	function	returns natural log base raised to the x power
SQRT(x)	function	returns square root of x
SQR(x)	function	returns the square of x
MIN( )	function	returns the minimum value of the operands
MAX( )	function	returns the maximum value of the operands
SIZEOF(x)	function	returns the number of bytes required for a value of the type of a REAL, which is always 8

The type REAL represents floating point data. Variables of this type will occupy eight bytes of memory and will be aligned on a double word boundary. All REAL arithmetic is done using 370 long floating point. See section 5.2.1 on page 27 for implicit type conversions.

The type REAL has restrictions that other scalar types do not have. You may not take a subrange of REAL nor index an array by REAL. The predefined functions SUCC, PRED, ORD, HIGHEST and LOWEST are not defined for type REAL.

5.6 THE ARRAY TYPE

Syntax:

array-type:

```
──┬──> packed ──┐
  │<────────────┘
  └──────> array [ ──┬──>{index-type}──┬──> ] of ──>{type} ────────>
                     └<────── , <──────┘
```

index-type:

```
    ┌──>{enumerated-scalar-type}──>┐
────┼──>{id:scalar-type}──────────>┼──────────────────────────────>
    └──>{subrange-scalar-type}────>┘
```

The array type defines a list of homogeneous elements; each element is paired with one value of the index. An element of the array is selected by a subscript. The number of elements in the array is the number of values potentially assumable by the index. Each element of the array is of the same type, which is called the element type of the array. Entire arrays may be assigned if they are of the same type.

Pascal/VS uses square brackets, '[' and ']', in the declaration of arrays. Because these symbols are not directly available on many I/O devices, the symbols '(.' and '.)' may be used as an equivalent to square brackets.

Pascal/VS will align each element of the array, if necessary, to make each element fall on an appropriate boundary. A packed array will not observe the boundary requirements of its elements. Elements of packed arrays may not be passed as var parameters to routines.

An array which is defined with more than one index is said to be a multi-dimensional array. A multi-dimensional array is exactly equivalent to an array of arrays. In short, an array definition of the form

```
array[i,j,... ] of T
```

is an abbreviated form of

```
array[i] of
  array[j] of
    ... T
```

where i and j are scalar type definitions. Thus, the first and second type declarations in the example below are alternatives to the same structure.

```
type

  MATRIX = array[ 1..10, 1..10 ] of
             REAL;

  MATRIXO = array[ 1..10 ] of
              array[ 1..10 ] of
                REAL;

  ABLE = array[BOOLEAN] of INTEGER;

  COLOR     = (RED, YELLOW, BLUE);

  INTENSITY = packed array[COLOR]
                of REAL;

  ALFA = packed array[ 1..ALFALEN] of
           CHAR;
```

Examples of Array Declarations

There are two procedures available for conversion between a packed array and a similar but unpacked array. The predefined procedures PACK (section 11.2.1) and UNPACK (section 11.2.2) are provided for this purpose.

5.6.1 ARRAY SUBSCRIPTING

Array subscripting is performed by placing an expression in square brackets following an array variable. The expression must be of a type that is compatible with the index type and evaluate to one of the values of the index. (See section 7.1 on page 57). The index may be any scalar type except REAL.

```
var
  M        : MATRIX;
  HUE      : INTENSITY;

begin

  (* assign ten element array      *)
  M[1]          := M[2];

  (* assign one element of a two   *)
  (* dimensional array two ways    *)
  M[1,1]        := 3.14159;
  M[1][1]       := 3.14159;

  (* this is a reddish orange      *)
  HUE[RED]      := 0.7;
  HUE[YELLOW]   := 0.3;
  HUE[BLUE]     := 0.0;
```

Examples of Array Indexing

5.7 THE RECORD TYPE

Syntax:

```
record-type:

──────┬──────────────────┬──> record ──>{field-list}──> end ──────────────>
      └──> packed ──>────┘

field-list:

──────┬──>{fixed-part}──┬─────> ; ──┬──>{variant-part}──┬───┬──> ; ──┬──>
      │                 └───────────┤                   │   └───────>┘
      └────────────────────────────>┘                   │
                        └──────────────────────────────>┘

fixed-part:

──────┬────┬──┬──>{field}──┬────┬──> : ──>{type}──┬──────────────────────>
      │    │  └<── , <─────┘    │                 │
+     │    └───────────────────>┘                 │
      └<──────────────── ; <──────────────────────┘

variant-part:
+
──> case ──┬──>{field}──> : ──┬─────────────────────>┐
           └──────────────────┴──>{id:type}──────────┴──> of ──>┐
                                                                 │
   ┌<────────────────────────────────────────────────────────────┘
   │
   └──┬──┬──>{range}──┬──> : ( ──┬──>{field-list}──┬──> ) ──┬──────────>
      │  └<── , <─────┘          └────────────────>┘        │
      └<──────────────────── ; <─────────────────────────────┘

field:

──>{id}──┬──────────────────────────────────────┬──────────────────────────>
+        └───> ( ───>{constant-expr}───> ) ───>┘

range:

+ ──>{constant-expr}──┬───────────────────────────────────┬────────────────>
+                     └───> .. ───>{constant-expr}───>────┘
```

A record is a data structure which is composed of heterogeneous components; each element may be of a different type. Components of a record are called <u>fields</u>.

5.7.1 NAMING OF A FIELD

A field is referred to by the name of the field. The scope of the identifiers used as names is the record type itself. That is, every field name within a record must be unique, even if that name appears in a variant part.

+ A field of a record need not be named;
+ that is, the field identifier may be
+ missing. In such a case, the field only
+ serves as padding; it can not be refer-
+ enced.

```
type
  REC =    record
           A,
           B  : INTEGER;
              : CHAR;        (*unnamed*)
           C  : CHAR
           end;

  DATE =   record
           DAY   : 1..31;
           MONTH : 1..12;
           YEAR  : 1900..2100
           end;

  PERSON = record
           LAST_NAME,
           FIRST_NAME     : ALFA;
           MIDDLE_INITIAL : CHAR;
           AGE            : 0..99;
           EMPLOYED       : BOOLEAN
           end;
```

Simple Record Declarations

5.7.2 FIXED PART

The fixed part of a record is a series of fields common to all variables of a given record type. The fixed part, if present, is always before the variant part.

5.7.3 VARIANT PART

The variant part of a record permits the defining of an alternative structure to the record. The record structure adopts one of the variants at a time.

The variant part of a record is denoted with the case symbol. A tag field identifier may follow. The value of this field indicates which variant is intended to be active.

The tag field is a field in the fixed part of the record. When the tag field is followed by a type identifier, then the tag field defines a new field within the record.

+ If the type identifier is missing, then the tag field name must be one which was previously defined within the record. This allows you to place the tag field anywhere in the fixed part of the record.

A variant part of a record need not have a tag field at all. In this case, only a type identifier is specified in the case construct. You still refer to the variant fields by their names but it is your responsibility to keep track of which variant is 'active' (i.e. contains valid data) during execution.

In short, tag fields may be defined in the following ways:

- "case I : INTEGER of" results in I being a tag field of type INTEGER.
- "case INTEGER of" means no tag field is present, the variants are denoted by integer values in the variant declaration.
- "case I: of" means that I is the tag field and it must have been declared in the fixed part, the type of I is as given in the field definition of I.

The following examples illustrate the three tag fields in complete record definitions.

```
type

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

  COORDINATES =
                    (* fixed part:     *)
    record
      X,Y      : REAL;
      AREA     : REAL;
      case S   : SHAPE of
                    (* variant part:   *)
      TRIANGLE:
            (SIDE : REAL;
         BASE : REAL);

      RECTANGLE:
         (SIDEA,SIDEB : REAL);

      SQUARE:
         (EDGE : REAL);

      CIRCLE:
         (RADIUS : REAL)
    end;
```

A Record With a Variant Part

The record defined as COORDINATES in the example above contains a variant part. The tag field is S, its type is SHAPE, and its value (whether TRIANGLE, RECTANGLE, SQUARE, or CIRCLE) indicates which variant is in effect. The fields SIDE, SIDEA, EDGE, and RADIUS would all occupy the same offset within the record. The following diagram illustrates how the record would look in storage.

fixed part:	
	X
	Y
	AREA
tag field:	S

variant part:

SIDE	SIDEA	EDGE	RADIUS
BASE	SIDEB		

Each column in the variant represents one alternative for the variant.

If you preferred the tag field to be the first field instead of the fourth, you could define it as follows:

```
COORDINATES =
  record
    S       : SHAPE;
    X,Y     : REAL;
    AREA    : REAL;
    case S  : of
                (* variant part:    *)
    TRIANGLE:
       (SIDE : REAL;
        BASE : REAL);

    RECTANGLE:
       (SIDEA,SIDEB : REAL);

    SQUARE:
       (EDGE : REAL);

    CIRCLE:
       (RADIUS : REAL)
  end;
```

Record with Back Reference Tag Field

If you preferred the tag field to be absent altogether you could define the record as follows:

```
COORDINATES =
  record
    X,Y     : REAL;
    AREA    : REAL;
    case SHAPE of
                (* variant part:    *)
    TRIANGLE:
       (SIDE : REAL;
        BASE : REAL);

    RECTANGLE:
       (SIDEA,SIDEB : REAL);

    SQUARE:
       (EDGE : REAL);

    CIRCLE:
       (RADIUS : REAL)
  end;
```

Record Variant with No Tag Field

5.7.4 PACKED RECORDS

The fields in a record are normally assigned offsets sequentially, padding where necessary for boundary alignment. In packed records, however, no such padding is done. This may save storage within the record, but may degrade performance of the program. Fields of packed records may not be passed as var parameters to a routine.

5.7.5 OFFSET QUALIFICATION OF FIELDS

Pascal/VS provides you a method of forcing the fields of a record to begin at a specified byte offset in the record. A field name may be followed by a integer constant expression enclosed in parentheses which represents the byte offset within the record that the field is to represent. All fields so specified must be in consecutive order according to offsets. If the offset is not specified, the field will be assigned the next offset that is required for boundary alignment. If an offset specification attempts to assign an incorrect boundary for a field and the record is not packed, a compile time error will be raised.

As an example of offset qualified fields within a record, consider a large control block of 100 bytes, in which four fields at various offsets need to be referenced.

byte displacement	information
0	field A (integer)
36	field B (8 chars)
80	field C (4 flags)
92	field D (integer)

The control block might be represented in Pascal/VS as follows:

```
  type
    FLAGS = set of
              (F1,F2,F3,F4);
    PADDING = packed array[1..4] of
                CHAR;
    CB     = packed record
              A      : INTEGER;
              B(36)  : ALFA;
              C(80)  : FLAGS;
              D(92)  : INTEGER;
                     : PADDING
            end;
  var
    BLOCK : CB;
```

A Record with Offset Qualified Fields

5.8 THE SET TYPE

```
Syntax:

set-type:

 ──┬──────────────────┬──> set of ──>{base-scalar-type}──────────────>
   └──> packed ──>────┘

base-scalar-type:

    ┌──>{enumerated-scalar-type}──>┐
 ───┼──>{id:scalar-type}──────────>┼─────────────────────────────────>
    └──>{subrange-scalar-type}─────>┘
```

A variable whose type is a set may contain any combination of values taken from the <u>base scalar type</u>. A value is either in the set or it is not in.

Note: Pascal/VS sets can be used in many of the same ways as bit strings (which often tend to be machine dependent). For example, a set operation such as intersection (the operator is '*') is the same as taking the 'boolean and' of two bit strings.

```
type
  CHARS       = set of CHAR;
  DAYSOFMON   = packed set of 1..31;
  DAYSOFWEEK  = set of MONDAY..FRIDAY;
  FLAGS       = set of
                  (A,B,C,D,E,F,G,H);
```

Set Declarations

The following table describes the operations that apply to the variables of a set type.

Set Operators

operation	form	description
¬	unary	returns the complement of the operand
=	binary	compares for equality
<> or ¬=	binary	compares for inequality
<=	binary	returns TRUE if first operand is subset of second operand
>=	binary	returns TRUE if first operand is superset of second operand
in	binary	TRUE if first operand (a scalar) is a member in the set represented by the second operand
+	binary	forms the union of two sets
*	binary	forms the intersection of two sets
-	binary	forms the difference between two sets
&&	binary	forms an 'exclusive' union of two sets
SIZEOF(x)	function	returns the number of bytes required for a value of the type of x

Set union produces a set which contains all of the elements which are members of the two operands. Set intersection produces the set that contains only the elements common to both sets. Set difference produces the set which includes all elements from the left operand except those elements which are members of the right operand. Set exclusive union produces the set which contains all elements from the two operands except the elements which are common to both operands. The in operator tests for membership of a scalar within a set; if the scalar is not a permissible value of the set and checking is enabled, then a runtime diagnostic will result.

The storage and alignment required for a set variable is dependent on the scalar type on which the set is based. The amount of storage required for a packed set will be the minimum number of bytes needed to contain the largest member of the set. Given a set definition:

```
type
  S = set of BASE;

where BASE is a scalar type which is
        not a subrange
```

the ordinal value of the largest member M which is in the set is:

```
M := ORD(HIGHEST(BASE))
```

The following table indicates the mapping of a set variable as a function of M.

Range of M	Size in Bytes	Alignment
0 <= M <= 7	1	BYTE
8 <= M <= 15	2	HALFWORD
16 <= M <= 23	3	FULLWORD
24 <= M <= 31	4	FULLWORD
32 <= M <= 255	(M+7) div 8	BYTE

Unpacked sets based upon integer (or subranges of integers) will occupy 32 bytes. The maximum value of a member of a set of integer may not exceed 255.

The storage is the same for all unpacked sets of subranges of a base scalar type. The following illustrates this point.

```
Given:
  type
    T   =  set of t;
    S   =  set of s;

Where:
  t is a subrange of s.
```

The types T and S have identical storage mappings.

5.9 THE FILE TYPE

```
Syntax:

file-type:

---> file of --->{type}----------------------------------------->
```

All input and output in Pascal/VS use the file type. A file is a structure consisting of a sequence of components where each component is of the same type. Variables of this type reference the components with pointers called file pointers. A file pointer could be thought of as a pointer into an input/output buffer.

The association of a file variable to an actual file of the system is implementation dependent and will not be described in this manual. Refer to the Programmer's Guide for this information.

```
type
   TEXT = file of CHAR;
   LINE = file of
            packed array[1..80] of
              CHAR;
   PFILE = file of
             record
               NAME: packed
                 array[1..25] of
                   CHAR;
               PERSON_NO:INTEGER;
               DATE_EMPLOYED:DATE;
               WEEKLY_SALARY:INTEGER
             end;
```

File Declarations

You access the file through predefined procedures and functions. (see section 10.0 on page 93) They are:

- GET (Section 10.6)
- PUT (Section 10.7)
- EOF (Section 10.8)
- EOLN (Section 10.13)
- RESET (Section 10.1)
- REWRITE (Section 10.2)
- + INTERACTIVE (Section 10.3)
- + OPEN (Section 10.4)
- + CLOSE (Section 10.5)
- READ (Section 10.9)
- WRITE (Section 10.11)

OUTPUT is predefined as a TEXT file variable. This is the file which will receive Pascal/VS execution time diagnostics.

Pascal/VS enforces the following restrictions on the file type:

1. A file may be passed by var or passed by const, but never by value to a procedure or function.
2. A file may not be an element of an array.
3. A file may not be a field of a record.
4. A file may not be contained within a file.

5.10 PREDEFINED STRUCTURE TYPES

5.10.1 THE TYPE STRING

```
Syntax:

string-type:

---> STRING ---T---> ( --->{constant-expr}---> ) ---T----------------------->
               L------------------------------------>J
```

The type STRING is defined as a 'packed array[1..n] of CHAR' whose length varies at execution time up to a compile time specified maximum. The length of the array is obtained during execution by the LENGTH function (section 11.6.1). The length is managed implicitly by the operators and functions which apply to STRINGs. The length of a STRING variable is determined when the variable is assigned. By definition, string constants belong to the type STRING.

STRING variables may be subscripted to retrieve individual characters. Upon subscripting, the variable behaves as though it were declared as a 'packed array[1..n] of CHAR', where n is the current length of the STRING.

The constant expression which follows the STRING qualifier in the type definition is the maximum length that the string may obtain and must be in the range of 1 to 255. If the value is not specified, the maximum length of 255 is assumed.

Any variable of a STRING type is compatible with any other variable of a STRING type; that is, the maximum length field of a type definition has no bearing in type compatibility tests.

Implicit conversion is performed when assigning a STRING to a packed array[1..n] of CHAR'. All other conversion must be done explicitly.

The assignment of one string to another may cause a run time error if the actual length of the source string is greater then the maximum length of the target. Pascal/VS will never truncate implicitly.

```
function GETCHAR(
         const S   : STRING;
               IDX : INTEGER) : CHAR;
  begin
    GETCHAR := S[IDX]
  end;
  .
  .
var
  S1: STRING(10);
  S2: STRING(5);
  C:  CHAR;
begin
  S1 := 'MESSAGE:';
  C  := GETCHAR(S1,4);
  (* C assigned 'S' *)
    .
    .
  S2 := 'FIVE';
  C  := GETCHAR(S2,2);
  (* C assigned 'I' *)
end;
```

Usage of STRING Variables

The following table describes the operations and predefined functions that apply to the variables of type STRING.

STRING		
operation	form	description
=	binary	compares for equality+
<> or ¬=	binary	compares for inequality+
<	binary	compares for left less than right+*
<=	binary	compares for left less than or equal to right+*
>=	binary	compares for left greater than or equal to right+*
>	binary	compares for left greater than right+*
||	binary	catenates the operands
LENGTH	function	returns the length of the STRING (section 11.6.1, page 124)
LBOUND	function	returns the value 1, STRINGS always have a lower bound of one (section 11.3.1, page 110)
HBOUND	function	returns the declared maximum number of elements of the string (section 11.3.2, page 111)
SUBSTR	function	returns a specified portion of a STRING (section 11.6.2, page 124)
DELETE	function	returns a STRING with a portion removed (section 11.6.2, page 124)
TRIM	function	returns a STRING with trailing blanks removed (section 11.6.4, page 125)
LTRIM	function	returns a STRING with leading blanks removed (section 11.6.5, page 126)
COMPRESS	function	returns a STRING with multiple blanks removed (section 11.6.6, page 126)
INDEX	function	locates a STRING in another STRING (section 11.6.7, page 127)
SIZEOF(x)	function	returns the number of bytes required for a value of the type of x

+ If two STRINGs being compared are of different lengths, the shorter is assumed to be padded with blanks on the right until the lengths match.
* Relative magnitude of two strings is based upon the collating sequence of EBCDIC.

STRING Conversions with Relational Operators

LEFT OPERAND	RIGHT OPERAND: relational operations	CHAR	packed array[1..n] of CHAR	STRING
	CHAR	allowed	not permitted	use STR on the CHAR
	packed array[1..n] of CHAR	not permitted	okay if the types are compatible	use STR on the array
	STRING	use STR on the CHAR	use STR on the array	allowed

STRING Conversions on Assignment				
		FROM		
	assignment	CHAR	packed array[1..n] of CHAR	STRING
T O	CHAR	allowed	not permitted	use string indexing to obtain char
	packed array[1..n] of CHAR	not permitted	okay if the types are compatible	okay, STRING is converted. If truncation is required, then an error results.
	STRING	use STR to convert CHAR to a STRING	use STR to convert array to a STRING	allowed

+ ## 5.10.2 THE TYPE ALFA

+ The standard type ALFA is defined as:

```
+  const
+    ALFALEN = 8;

+  type
+    ALFA = packed
+              array[1..ALFALEN] of
+                CHAR;
```

+ Any 'packed array[1..n] of CHAR',
+ including ALFA, may be converted to
+ type STRING by the predefined function
+ STR. The following table describes the
+ operations and predefined functions
+ that apply to the variables of the pre-
+ defined type ALFA.

ALFA		
operation	form	description
=	binary	compares for equality
<> or ¬=	binary	compares for inequality
<	binary	compares for left less than right
<=	binary	compares for left less than or equal to right
>=	binary	compares for left greater than or equal to right
>	binary	compares for left greater than right
STR(x)	function	converts the ALFA to a STRING
SIZEOF(x)	function	returns the number of bytes required for a value of the type of an ALFA, which is always 8

+ 5.10.3 THE TYPE ALPHA

+ The standard type ALPHA is defined as:

```
+  const
+    ALPHALEN = 16;

+  type
+    ALPHA = packed
+                  array[1..ALPHALEN] of
+                    CHAR;
```

+ Any 'packed array[1..n] of CHAR', including ALPHA, may be converted to type STRING by the predefined function STR. The following table describes the operations and predefined functions that apply to the variables of the predefined type ALPHA.

ALPHA		
operation	form	description
=	binary	compares for equality
<> or ¬=	binary	compares for inequality
<	binary	compares for left less than right
<=	binary	compares for left less than or equal to right
>=	binary	compares for left greater than or equal to right
>	binary	compares for left greater than right
STR(x)	function	converts the ALPHA to a STRING
SIZEOF(x)	function	returns the number of bytes required for a value of the type of an ALPHA, which is always 16

5.10.4 THE TYPE TEXT

The standard type TEXT is defined as:

```
type
  TEXT     = file of CHAR;
```

In addition to the predefined procedures to do input and output, Pascal/VS defines several procedures which operate only on files of type TEXT. These procedures perform character to internal representation (EBCDIC) conversions and gives you some control over output field lengths. The predefied routines that may be used on TEXT files are:

- GET (Section 10.6)
- PUT (Section 10.7)
- EOF (Section 10.8)
- EOLN (Section 10.13)
- RESET (Section 10.1)
- REWRITE (Section 10.2)
- READ (Section 10.9)
- READLN (Section 10.9)
- WRITE (Section 10.11)
- WRITELN (Section 10.11)
- PAGE (Section 12.5)
- INTERACTIVE (Section 10.3)
- OPEN (Section 10.4)
- CLOSE (Section 10.5)
- COLS (Section 10.15)

Pascal/VS predefines two TEXT variables named OUTPUT and INPUT. You may use these files without declaring them in your program.

5.11 THE POINTER TYPE

```
Syntax:

pointer-type:

---> -> --->{id:type}------------------------------------------->
```

Pascal/VS allows variables to be created during program execution under your explicit control. These variables, which are called dynamic variables, are generated by the predefined procedure NEW. NEW creates a new variable of the appropriate type and assigns its address to the argument of NEW. You must explicitly deallocate a dynamic variable; the predefined procedures DISPOSE and RELEASE are provided for this purpose.

\+ Dynamic variables are created in an area of storage called a heap. A new heap is created with the MARK predefined procedure; a heap is released with the RELEASE predefined procedure. A initial heap is allocated by Pascal/VS. All variables that were allocated in a heap are deallocated when the heap is released. An attempt to use a dynamic variable that has been deallocated (either via DISPOSE or RELEASE) is an error. Refer to section 11.1.1, page 106 for details on MARK, RELEASE, DISPOSE and NEW.

Pascal/VS pointers are constrained to point to a particular type. This means that on declaration of a pointer, you must specify the type of the dynamic variable that will be generated by NEW or referenced.

Pascal/VS defines the named constant nil as the value of a pointer which does not point to any dynamic variable (empty pointer). Nil is type compatible to every pointer type.

The only operators that can be applied to variables of pointer type are the test for equality and inequality. The predefined function ORD may be applied to a pointer variable; the result of the function is an integer value which is equal to the address of the dynamic variable referenced by the pointer. There is no function in Pascal/VS to convert an integer into a pointer.

```
type
  PTR = -> ELEMENT;
  ELEMENT = record
              PARENT : PTR;
              CHILD  : PTR;
              SIBLING: PTR
            end;
```

A Pointer Declaration

This example illustrates a data types that can be used to build a tree. With this structure the parent node contains a pointer to the eldest child, the eldest points to the next sibling who points to the next, and so forth.

In the above example type ELEMENT was used before it was declared. Referencing an identifier prior to its declaration is generally not permitted in Pascal/VS. However, a type identifier which is used as the base type to a pointer declaration is an exception to this rule.

5.12 STORAGE, PACKING, AND ALIGNMENT

For each variable declared with a particular type, Pascal/VS allocates a specific amount of storage on a specific alignment boundary. The Programmer's Guide describes implementation requirements and defaults.

Pascal/VS provides the packed record feature in which all boundary alignment is suppressed. Fields of a packed record are allocated on the next byte, ignoring alignment requirements.

Packed data occupies less space and is more compact but may increase the execution time of the program. Moreover, a field of a packed record or an element of a packed array may not be passed by read/write reference (var) to a routine.

6.0 ROUTINES

Syntax:

routine-dcl:

```
  ────┬──>{procedure-heading}──────────────────>┐
      └──>{function-heading}───────────────────>┤
                                                 │
     ┌<──────────────── ; <─────────────────────┘
     │
+    ├---> EXTERNAL ---> ; -------------------------------->┐
+    ├---> FORTRAN ----> ; -------------------------------->┤
+    ├---> ENTRY ---> ; --->┐                               │
+    ├<---------------------┘                               │
     ├──> FORWARD ──> ; ───────────────────────────────────>┤
     │                                                      │
     ├<──────────────────┐                                  │
     ├──>{declaration}──>┘                                  │
     │                                                      │
     └──>{compound-statement}──> ; ─────────────────────────┴──────────>
```

procedure-heading:

```
──> procedure ──>{id}──>{formal-parameters}──────────────────────────>
```

function-heading:

```
──> function ──>{id}──>{formal-parameters}──> : ──>{id:type}──────────>
```

formal-parameters:

```
────┬───> ( ──┬──>{formal}──┬──> ) ──┬──────────────────────────────>
    │         └<──── ; <────┘        │
    └───────────────────────────────>┘
```

formal:

```
────┬──┬──> var ────┬──┬──{id}──┬──> : ──>{id:type}────────┬──────>
    │  └---> const -->┘  └<── , <──┘                         │
    │                                                        │
    ├──┬──{id}──┬──> : ───>{id:type}────────────────────────>┤
    │  └<── , <──┘                                           │
    │                                                        │
    ├──>{procedure-heading}─────────────────────────────────>┤
    │                                                        │
    └──>{function-heading}──────────────────────────────────>┘
```

There are two categories of routines: procedures and functions. Procedures should be thought of as adding new statements to the language. These new statements effectively increase the language to a superset language containing statements tailored to your specialized needs. Functions should also be thought of as increasing the flexibility of the language: functions add to your ability to express data transformation in expressions.

Routines can return data to the caller by altering the var parameters or by assigning to variables that are common to both the invoker and the invoked routine. In addition, functions also return a value to the invoker upon return from the function.

6.1 ROUTINE DECLARATION

Routines must be declared prior to their use. The routine declaration consists of the routine heading, declarations and one compound statement.

The heading defines the name of the routine and binds the formal parameters to the routine. The heading of a function declaration also binds the function name to the type of value returned by the function. Formal parameters specify data that is to be passed to the routine when it is invoked. The declarations are described in chapter 4. The compound statement will be executed when the routine is invoked.

6.2 ROUTINE PARAMETERS

Formal parameters are bound to the routine when the routine is defined. The formal parameters define what kind of data may be passed to the routine when it is invoked. These parameters also specify how the data will be passed.

When the routine is invoked, a parameter list is built. At the point of invocation the parameters are called the *actual parameters*.

Pascal/VS permits parameters to be passed in following ways:

- pass by value
- pass by read/write reference (var)
- pass by read only reference (const)
- formal routine parameter

6.2.1 PASS BY VALUE PARAMETERS

Pass by value parameters can be thought of as local variables that are initialized by the caller. The called routine may change the value of this kind of parameter but the change is never reflected back to the caller. Any expression, variable or constant (except of file type) may be passed with this mechanism.

6.2.2 PASS BY VAR PARAMETERS

This method is also called *pass by reference*. Parameters that are passed by var reflect modifications to the parameters back to the caller. Therefore you may use this parameter type as both an input and output parameter. The use of the var symbol in a parameter indicates that the parameter is to be passed by read/write reference. *Only variables may be passed by this mechanism; expressions and constants may not*. Also, fields of a packed record or elements of a packed array may not be passed as var parameters.

6.2.3 PASS BY CONST PARAMETERS

Parameters passed by const may not be altered by the called routine. Also you should not modify the actual parameter value while the call to the routine has not yet completed. If you attempt to alter the actual parameter while a it is being passed by const, the result is not defined. This method could be called *pass by read only reference*. The parameters appear to be constants from the called routine's point of view. Any expression, variable or constant may be passed by const (fields of a packed record and elements of a packed array may also be passed). The use of the "const" reserved word in a parameter indicates that the parameter is to be passed by this mechanism. With parameters which are structures (such as strings), passing by const is usually more efficient than passing by value.

6.2.4 FORMAL ROUTINE PARAMETERS

A procedure or function may be passed to a routine as a formal parameter. Within the called routine the formal parameter may be used as if it were a procedure or function.

6.3 ROUTINE COMPOSITION

There are five kinds of routine declarations:

- internal
- EXTERNAL
- FORTRAN
- ENTRY
- FORWARD

The directive used to identify each kind of declaration is shown in upper case above.

An *internal* routine may be invoked only from within the lexical scope that contains the routine definition.

An *EXTERNAL* routine can be invoked from within the lexical scope that contains the declaration but the routine body is defined outside the module. The formal parameters defined in the EXTERNAL routine declaration must match those in the module where the routine is defined, but this is the programmer's responsiblity. An EXTERNAL routine may refer to a Pascal/VS routine which is declared in another module or it may refer to code produced by other means

+ (such as assembler code).

+ A FORTRAN routine is similiar to an + EXTERNAL routine in that it specifies a + routine that is defined outside the + module being compiled. In addition, it + specifies that the routine is a FORTRAN + subprogram and that the conventions of + FORTRAN are to be used. If you pass a + literal constant to a FORTRAN subpro- + gram by CONST, then you must assure + that the FORTRAN subprogram does not + alter the contents of parameter. In + order to meet the requirements of + FORTRAN you must obey the following + restrictions:

+ - All parameters may be only var or + const parameters.

+ - If the routine is a function, it + may only return a scalar result + (including a REAL).

+ - Routines may not be passed.

+ - Multi-dimensional arrays are not + remapped to conform to FORTRAN + indexing, that is, an element of an + array A[n,m] in Pascal will be ele- + ment A(m,n) in FORTRAN.

+ An ENTRY routine declaration defines a + routine that can be invoked from anoth- + er module. An ENTRY routine can be + invoked from within the module in which + it is declared and from any module that + declares it as EXTERNAL.

+ ENTRY routines can only be declared at + the outermost nesting level. That is, + they must be declared directly within + the program module or directly within a + segment module; they can never be + declared within another routine.

A routine declared FORWARD is the means by which you can declare the routine heading before declaring the declarations and compound statement. The routine heading is declared followed by the symbol 'FORWARD'. This allows you to have a call to a routine prior to defining the routine's body. If two routines are to be mutually recursive and are at the same nesting level, one of the routines must be declared FORWARD.

To declare the body of the FORWARD routine, you declare the routine leaving off the formal parameter definition. A routine declared as an ENTRY routine may also be FORWARDed.

Notes:

1. Pascal/VS allows routines to be textually nested to a depth not greater than eight.

2. A routine must be declared before it can be referenced. This allows the compiler to assure the validity of a call by checking parameter compatiblity.

```
static
  C: CHAR;

function GETCHAR:CHAR;
  EXTERNAL;

procedure EXPR(var VAL: INTEGER);
  ENTRY; FORWARD;

procedure FACTOR(var VAL: INTEGER);
  ENTRY;
  begin
    C := GETCHAR;
    if C = '(' then
      begin
        C := GETCHAR;
        EXPR(VAL)
      end
    else
      .
      .
  end;

procedure EXPR (*var VAL: INTEGER*);
  begin
    FACTOR(VAL);
      .
      .
  end;
```

Examples of Routine Declarations

```
function CHARFOUND
   (const S: STRING;
          C: CHAR): BOOLEAN;
  var I: 1..255;
  begin
    for I := 1 to LENGTH(S) do
      if S[I] = C then
        begin
          CHARFOUND := TRUE;
          return
        end;
    CHARFOUND := FALSE;
  end;
```

Example of Const Parameter

6.4 FUNCTION RESULTS

A value is returned from a function by assigning the value to the name of the function prior to leaving the function. This value is inserted within the expression at the point of the call. The value must be assignment conformable to the type of the function.

If the function name is used on the right side of an assignment, it will be interpreted as a recursive call.

```
function FACTORIAL
          (X: INTEGER): INTEGER;
  begin
    if X <= 1 then
      FACTORIAL := 1
    else
      FACTORIAL := X * FACTORIAL(X-1)
  end;
```

Example of Recursive Function

6.5 PREDEFINED PROCEDURES AND FUNCTIONS

Pascal/VS predefines a number of procedures and functions that you may find valuable. Details of the predefined procedures and functions are given in section 10.0 beginning at page 93.

7.0 VARIABLES

```
Syntax:

variable:                                                      notes:
──>{id}──>┐
          ├<──────────────────────────────────────┐
          │                                        │
          ├──> [ ──┬──>{expr}──┬──> ] ──────────>┤   array reference
          │        └<──── , ───┘                   │
          ├──> . ──>{id:field}──────────────────>┤   field reference
          │                                        │
          ├──> -> ──────────────────────────────>┘   pointer reference
          │
          └────────────────────────────────────────────────────────────>
```

Pascal/VS divides variables into five classes depending on how they are declared:

- automatic (var variables)
- dynamic (pointer-qualified variables)
- \+ static (static variables)
- \+ external (def/ref variables)
- parameter (declared on a routine declaration)

A variable may be referenced in several ways depending on the variable's type. You may always refer to the entire variable by specifying its name. You may refer to a component of a structured variable by using the syntax shown in the syntax diagram.

If you simply specify the name of the variable, then you are referring to the entire variable. If that variable is declared as an array, then you are referring to the entire array. You may assign an entire array. Similarly, you may deal with record and set variables as units.

```
var
  LINE1,
  LINE2 : packed
            array[ 1..80 ] of
              CHAR;
   .
   .
 (* assign all 80 characters       *)
 (*    of the array                *)
 LINE1 := LINE2;
```

Using Variables in their entirety

7.1 ARRAY REFERENCING

An element of an array is selected by placing an indexing expression enclosed within square brackets, after the name of the array. The indexing expression must be of the same type as declared on the corresponding array index definition.

A multi-dimensional array may be referenced as an array of arrays. For example, let variable A be declared as follows:

```
A: array [a..b,c..d] of T
```

As explained in section 5.6, this declaration is exactly equivalent to:

```
A: array [a..b] of
     array [c..d] of T
```

A reference of the form A[I] would be a variable of type:

```
array [c..d] of T
```

and would represent a single row in array A. A reference of the form A[I][J] would be a variable of type T and would represent the Jth element of the Ith row of array A. This latter reference would customarily be abbreviated as

```
A[I,J]
```

Any array reference with two or more subscript indicies can be abbreviated by writing the subscripts in a comma separated list. That is, A[I][J]... could be written as A[I,J,...].

If the '%CHECK SUBSCRIPT' option is enabled, the index expression will be checked at execution time to make sure its value does not lie outside of the subscript range of the array. An execution time error diagnostic will occur if the value lies outside of the prescribed range. (For a description of the CHECK feature see section 12.2 on page 134)

```
A[12]
A[I]
A[ I+J ]
DECK[ CARD-FIFTY ]
MATRIX[ ROW[I], COLUMN[J] ]
```

Subscripted Variables

7.2 FIELD REFERENCING

A field of a record is selected by following the record variable by a period and by the name of the field to be referenced.

```
var
  PERSON:
    record
      FIRST_NAME,
      LAST_NAME: STRING(15);
    end;

  DATE:
    record
      DAY: 1..31;
      MONTH: 1..12;
      YEAR: 1900..2000
    end;

  DECK:
    array[1..52] of
      record
        CARD: 1..13;
        SUIT:
          (SPADE, HEART,
           DIAMOND, CLUB)
      end;
        .
        .
        .
PERSON.LAST_NAME := 'SMITH';
DATE.YEAR := 1978;
DECK[ I ].CARD := 2;
DECK[ I ].SUIT := S;
```

Field Referencing Examples

7.3 POINTER REFERENCING

A dynamic variable is created by the predefined procedure NEW or by an implementation provided routine which assigns an address to a pointer variable. You may refer either to the pointer or to the dynamic variable; referencing the dynamic variable requires using the pointer notation.

For example

```
var P : -> R;

  P    refers to the pointer
  P->  refers to the dynamic variable
```

If the '%CHECK POINTER' option is enabled, any attempt to reference a pointer that has not been assigned the address of an allocated variable will result in an execution time error diagnostic. (For a description of the CHECK feature see section 12.2 on page 134)

```
type
  INFO = record
           AGE:  1..99;
           WEIGHT: 1..400;
         end;

   FAMILY =
     record
       FATHER,
       MOTHER,
       SELF: ->INFO;
       KIDS: 0..10
     end;

var
  FAMILY_POINTER : ->FAMILY
        .
        .
NEW(FAMILY_POINTER);
FAMILY_POINTER->.KIDS := 2;
NEW(FAMILY_POINTER->.FATHER);
FAMILY_POINTER->.FATHER->.AGE := 35;
        .
        .
```

Pointer Referencing Examples

7.4 FILE REFERENCING

A component of a file is selected from the file buffer by a pointer notation. The file variable is assigned by using the predefined procedures GET and PUT.

Each use of these procedures moves the current component to the output file (PUT) or assigns a new component from the input file (GET). (For a description of GET and PUT, see sections 10.6 and 10.7.)

If the '%CHECK POINTER' option is enabled, any attempt to reference a file pointer which has no value will result in an execution time error diagnostic. (For a description of the CHECK feature see section 12.2 on page 134)

```
var
  INPUT     : TEXT;
  OUTPUT    : TEXT;
  LINE1     : array [1..80] of CHAR;
  .
  .
(* scan off blanks                   *)
(*   from a file of CHAR             *)
GET(INPUT);
while INPUT-> = ' ' do
  GET(INPUT);


(* transfer a line to the            *)
(*   OUTPUT file                     *)
for I := 1 to 80 do
  begin
       OUTPUT-> := LINE1[I];
    PUT(OUTPUT)
  end;
```

File Referencing Examples

8.0 EXPRESSIONS

Syntax:

constant-expr:
expr:

```
--->{simple-expression}--+------------------------------------------+--->
                         +--> =  ---+-->{simple-expression}--->-----+
                         +--> <> -->|
                         +--> <  -->|
                         +--> <= -->|
                         +--> >= -->|
                         +--> >  -->|
                         +--> in -->+
```

simple-expression:

```
-----+----------+------+---->{term}--+---------------------------------->
     +--> + --->|      |             +----> +  --->+
     +--> - --->+      |             +----> -  --->|
                       |             +----> && --->|
                       |             +----> |  --->|
                       +<------------------------------+
```

term:

```
----+----->{factor}--+---------------------------------------------->
    |                +---> *   ----->+
    |                +---> /   ----->|
    |                +---> div ----->|
    |                +---> mod ----->|
    |                +---> >>  ----->|
    |                +---> <<  ----->|
    |                +---> ||  ----->|
    |                +---> &   ----->|
    +<-----------------------------------+
```

factor:

```
----+----->{function-call}----------------------------+--------------->
    +----->{variable}-------------------------------->|
    +----->{set-factor}------------------------------>|
    +-----> ( --->{expr}---> ) ---------------------->|
    +----->{structured-constant}--------------------->|
    +-----> not --->{factor}------------------------->|
    +----->{unsigned-constant}----------------------->+
```

Pascal/VS expressions are similar in function and form to expressions found in other high level programming languages. Expressions permit you to combine data according to specific computational rules. The type of computation to be performed is directed by operators which are grouped into four classes according to precedence:

- the not operator (highest)
- the multiplying operators
- the adding operators
- the relational operators (lowest)

An expression is evaluated by performing the operators of highest precedence first, operators of the next precedence second and so forth. Operators of equal precedence are performed in a left to right order. If an operator has an operand which is a parenthesized sub-expression, the sub-expression is evaluated prior to applying the operator.

The operands of an expression may be evaluated in either order; that is, you should not expect the left operand of dyadic operator to be evaluated before the right operand. If either operand changes a global variable through a function call, and if the other operand uses that value, then the value used is not specified to be the updated value. The only exception is in boolean expressions involving the logical operations of 'and' (&) and 'or' (|); for these operations the right operand will not be evalauted if the result can be determined from the left operand. See section 8.3 on page 67.

Examples of Expressions

Assume the following declarations:

```
const
  ACME      = 'acme';

type
  COLOR     = (RED, YELLOW, BLUE);
  SHADE     = set of COLOR;
  DAYS      = (SUN, MON, TUES, WED, THUR, FRI, SAT);
  MONTHS    = (JAN, FEB, MAR, APR, MAY, JUN,
               JUL, AUG, SEP, OCT, NOV, DEC);

var
  A_COLOR   : COLOR;
  A_SET     : SHADE;
  BOOL      : BOOLEAN;
  I,
  J         : INTEGER;
```

factors:

```
I                         variable
15                        unsigned constant
(I*8+J)                   parenthetical expression
[ RED ]                   set of one element
[ ]                       empty set
ODD(I*J)                  function call
not BOOL                  complement expression
COLOR( 1 )                scalar type converter
ACME                      constant reference
```

terms:

```
I                         factor
I * J                     multiplication
I div J                   integer division
ACME || ' TRUCKING'       catenation
A_SET * [ RED ]           set intersection
I & 'FF00'X               logical and on integers
BOOL & ODD(I)             boolean and
```

simple expression:

```
I * J                     term
I + J                     addition
I | '80000000'X           logical or on integers
A_SET + [ BLUE ]          set union
- I                       unary minus on an integer
```

expression:

```
I + J                     simple expression
(I <= J) = (K > L)        relational operations
RED in A_SET              test for set inclusion
```

8.1 OPERATORS

	Multiplying Operators			
	operator	operation	operands	result
	*	multiplication	INTEGER REAL one REAL, one INTEGER	INTEGER REAL REAL
	/	real division	INTEGER REAL one REAL, one INTEGER	REAL REAL REAL
	div	integer division	INTEGER	INTEGER
	mod	modulo	INTEGER	INTEGER
	& (and)	boolean and	BOOLEAN	BOOLEAN
+	& (and)	logical and	INTEGER	INTEGER
	*	set intersection	set of t	set of t
+	||	string catenation	STRING	STRING
+	<<	logical left shift	INTEGER	INTEGER
+	>>	logical right shift	INTEGER	INTEGER

	Adding Operators			
	operator	operation	operands	result
	+	addition	INTEGER REAL one REAL, one INTEGER	INTEGER REAL REAL
	-	subtraction	INTEGER REAL one REAL, one INTEGER	INTEGER REAL REAL
	-	set difference	set of t	set of t
	| (or)	boolean or	BOOLEAN	BOOLEAN
+	| (or)	logical or	INTEGER	INTEGER
	+	set union	set of t	set of t
+	&& (xor)	exclusive or	BOOLEAN	BOOLEAN
+	&& (xor)	exclusive or	INTEGER	INTEGER
+	&& (xor)	'exclusive' union	set of t	set of t

The Not Operator			
operator	operation	operand	result
¬ (not)	boolean not	BOOLEAN	BOOLEAN
¬ (not)	logical one's complement	INTEGER	INTEGER
¬ (not)	set complement	set	set

Relational Operators			
operator	operation	operands	result
=	compare equal	any set, scalar type, pointer or string	BOOLEAN
<> (¬=)	compare not equal	any set, scalar type, pointer or string	BOOLEAN
<	compare less than	scalar type or string	BOOLEAN
<=	compare < or =	scalar type, string	BOOLEAN
<=	subset	set of t	BOOLEAN
>	compare greater	scalar type, string	BOOLEAN
>=	compare > or =	scalar type, string	BOOLEAN
>=	superset	set of t	BOOLEAN
in	set membership	t and set of t	BOOLEAN

+ 8.2 CONSTANT EXPRESSIONS

+ Constant expressions are expressions which can be evaluated by the compiler and replaced with a result at compile time. By its nature, a constant expression may not contain a reference to a variable or to a user-defined function. Constant expressions may appear in constant declarations.

+ The following predefined functions are permitted in constant expressions:

Function	Section	Page
- ABS	11.5.6	119
- CHR	11.4.2	113
- HIGHEST	11.3.4	112
- LENGTH	11.6.1	124
- LOWEST	11.3.3	111
- MAX	11.5.2	117
- MIN	11.5.1	117
- ORD	11.4.1	113
- PRED	11.5.3	118
- scalar conversion functions	11.4.3	114
- SIZEOF	11.3.5	112
- SUCC	11.5.4	118

constant expression	type
ORD('A')	INTEGER
SUCC(CHR('F0'X))	CHAR
256 div 2	INTEGER
'TOKEN'||STR(CHR(0))	STRING
'8000'X | '0001'X	INTEGER
['0'..'9']	set of CHAR
32768*2-1	INTEGER

Examples of Constant Expressions

8.3 BOOLEAN EXPRESSIONS

You should recognize that Pascal assigns the operations of "&" (and) and "|" a higher precedence than the relational operators. This means that the expression:

```
A<B & C<D
```

will be evaluated as :

```
(A < (B&C)) < D
```

Thus, it is advisable to use parenthesis when writing expressions of this sort.

Pascal/VS will optimize the evaluation of BOOLEAN expressions involving '&' (and) and '|' (or) such that the right operand of the expression will not be evaluated if the result of the operation can be determined by evaluating the left operand. For example, given that A, B, and C are boolean expressions and X is a boolean variable, the evaluation of

```
X := A or B or C
```

would be performed as

```
if A then
  X := TRUE
else
  if B then
    X := TRUE
  else
    X := C
```

The evaluation of

```
X := A and B and C
```

would be performed as

```
if ¬A then
  X := FALSE
else
  if ¬B then
    X := FALSE
  else
    X := C
```

The evaluation of the expression will always be left to right.

The following example demonstrates logic which depends on the conditional evaluation of the right operand of the "and" operator.

```
type
  RECPTR = ->REC;
  REC = record
          NAME: ALPHA;
          NEXT: RECPTR;
        end;

var
  P : RECPTR;
  LNAME : ALPHA;

begin
  ...
  while (P<>nil) and
        (P->.NAME <> LNAME)
  do
    P := P->.NEXT;
  ...
end;
```

Example of a BOOLEAN Expression that Depends on Order of Evaluation

Notes:

- If you use a function in the right operand of a boolean expression, then you must be aware that the function may not be evaluated. Further, you should note that relying on side-effects from functions is considered a bad programming practice.
- Not all Pascal compilers support this interpretation of BOOLEAN expressions. If you wish to assure portability between Pascal/VS and other Pascal implementations you should write the compound tests in a form that uses nested if-statements.

8.4 LOGICAL EXPRESSIONS

Many of the integer operators provided in Pascal/VS perform logical operations on their operands; that is, the operands are treated as unsigned strings of binary digits instead of signed arithmetic quantities. For example, if the integer value of -1 was used as an operand of a logical operation, it would be viewed as a string of binary digits with a hexadecimal value of 'FFFFFFFF'X.

The logical operations are defined to apply to 32 bit values. Such an operation on a subrange of an INTEGER could yield a result outside the subrange.

The following operators perform logical operations on integer operands:

- '&' (and) performs a bit-wise and of two integers.
- '|' (or) performs a bit-wise inclusive or.
- '&&' (xor) performs a bit-wise exclusive or.
- '¬' (not) performs a one's complement of an integer.
- '<<' shifts the left operand value left by the amount indicated in the right operand. Zeroes are shifted in from the right.
- '>>' shifts the left operand value right by the amount indicated in the right operand. Zeroes are shifted in from the left.

```
257 & 'FF'X      yields   1
2 | 4 | 8        yields   14
4 << 2           yields   16
-4 << 1          yields   -8
8 >> 1           yields   4
-8 >> 1          yields   '7FFFFFFC'X
'FFFF'X >> 3     yields   '1FFF'X
¬1 & 'FF'X       yields   'FE'X
¬0               yields   -1
'FF'X && 8       yields   'F7'X
```

Examples of Logical Operations

8.5 FUNCTION CALL

Syntax:

function-call:

```
——>{id:function}——>{actual-parameters}——————————————————————————>
```

actual-parameters:

```
——————┬——> ( ——┬———┬——>{expr}———┬———┬———> ) ——┬——————————————————>
      |        |   └<——— , <————┘   |          |
      |        └---------------------->┘          |
      └————————————————————————————————————————>┘
```

A function returns a value to the invoker. A call to a function passes the actual parameters to the corresponding formal parameters. Each actual parameter must be of a type that is conformable to the corresponding formal parameter. You may not pass a field of a packed record as a var parameter. You also may not pass an element of a packed array as a var parameter.

The parenthesis list may be dropped if the functions require no parameters. Also, you may use an empty set of parenthesis to show this.

```
var A,B,C: INTEGER;

function SUM
   (A,B: INTEGER): INTEGER;
  begin
    SUM := A+B
  end;
  ...
begin
  ...
  C := SUM(A,B) * 2
  ...
end;
```

Function Example

8.6 SCALAR CONVERSIONS

Pascal/VS predefines the function ORD that converts any scalar value into an integer. The scalar conversion functions convert an integer into a specified scalar type. An integer expression is converted to another scalar type by enclosing the expression within parentheses and prefixing it with the type identifier of the scalar type. If the operand is not in the range 0 .. ORD(HIGHEST(scalar type)), then a subrange error will result. The conversion is performed in such a way as to be the inverse of the ORD function. See section 11.4.3 on page 114.

The definition of any type identifier that specifies a scalar type (enumerated scalars or subranges) forms a scalar conversion function. By definition, the expression CHAR(x) is equivalent to CHR(x); INTEGER(x) is equivalent to x; and ORD(type(x)) is equivalent to x.

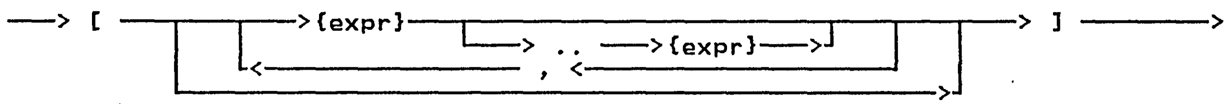
```
  type
    WEEK =
        (SUN,MON,TUE,WED,THU,FRI,SAT);
  var
    DAY: WEEK;
    .
    .
(*The following assigns SAT to DAY*)
  DAY := WEEK(6);
```

Scalar Conversion Functions

8.7 SET FACTORS

Syntax:

set-factor:



A set factor is used to compute a value of a set type within an expression.

The set factor is list of comma separated expressions within square brackets. Each expression must be of the same type; this type becomes the base scalar type of the set. If the set specifies INTEGER valued expressions, then there is an implementation restriction of 256 elements permitted in the set.

```
type
  DAYS    = set of
    (SUN,MON,TUES,WED,THU,FRI,SAT);
  CHARSET= set of CHAR;

var
  WORKDAYS,
  WEEKEND:     DAYS
  NONLETTERS: CHARSET;
  .
  .
  WORKDAYS := [MON..FRI];
  WEEKEND  := ¬ WORKDAYS;
  .
  .
  NONLETTERS :=
           ¬ ['a'..'z','A'..'Z'];
```

Set Factor

9.0 STATEMENTS

Syntax:

statement:

```
  ──┬──>{label}──> : ──>┐
    ├<──────────────────┘
+   ├--->{assert-statement}---------------------------------->┐
    ├───>{assignment-statement}──────────────────────────────>┤
    ├───>{case-statement}────────────────────────────────────>┤
    ├───>{compound-statement}────────────────────────────────>┤
+   ├--->{continue-statement}-------------------------------->┤
    ├───>{empty-statement}───────────────────────────────────>┤
    ├───>{for-statement}─────────────────────────────────────>┤
    ├───>{goto-statement}────────────────────────────────────>┤
    ├───>{if-statement}──────────────────────────────────────>┤
+   ├--->{leave-statement}----------------------------------->┤
    ├───>{procedure-call}────────────────────────────────────>┤
    ├───>{repeat-statement}──────────────────────────────────>┤
+   ├--->{return-statement}---------------------------------->┤
    ├───>{while-statement}───────────────────────────────────>┤
    └───>{with-statement}────────────────────────────────────>┴──────────>
```

Statements are your directions to perform specific operations based on the data. The statements are similar to those found in most high level programming languages.

+ 9.1 THE ASSERT STATEMENT

Syntax:

assert-statement:

```
---> assert --->{expr}------------------------------------------------------->
```

+ The assert statement is used to check for a specific condition and signal a runtime error if the condition is not met. The condition is specified by the expression which must evaluate to a BOOLEAN value. If the condition is not TRUE then the error is raised. The compiler may remove the statement from the program if it can be determined that the assertion is always true.

+ Example:

```
assert A >= B
```

The Assert Statement

9.2 THE ASSIGNMENT STATEMENT

Syntax:

assignment-statement:

```
──────┬──>{variable}──────────┬──> := ──>{expr}──────────────────────────────>
      └──>{id:function}───>───┘
```

The assignment statement is used to assign a value to a variable. This statement is composed of a reference to a variable followed by the assignment symbol (':='), followed by an expression which when evaluated is the new value. The variable must be conformable to the expression. The rules for expression conformability are given in section 5.2 on page 27.

You may make array assignments (assign one array to another array) or record assignments (assign one record to another). When doing this, the entire array or record is assigned.

A result is returned from a function by assigning the result to the function name prior to leaving the function. (See section 6.4 on page 55).

Pascal/VS will not permit the assignment of a value to a pass by const parameter.

Example:

```
type
  CARD = record
         SUIT : (SPADE,
                 HEART,
                 DIAMOND,
                 CLUB);
         RANK : 1..13
         end;

var
  X, Y, Z : REAL;

  LETTERS,
  DIGITS,
  LETTER_OR_DIGIT
     : set of CHAR;

  I, J, K : INTEGER;

  DECK : array[ 1..52 ] of
         CARD;
  ...
  X := Y*Z;
  LETTERS           := [ 'A' .. 'Z' ];
  DIGITS            := [ '0' .. '9' ];
  LETTER_OR_DIGIT   := LETTERS + DIGITS;
  DECK[ I ].SUIT    := HEART;
  DECK[ J ]         := DECK[ K ];
```

Assignment Statements

9.3 THE CASE STATEMENT

Syntax:

case-statement:

```
──> case ──>{expr}──> of ──>┐
 ┌<─────────────────────────┘
 │
 ├──────┬──┬──>{range}──┬───> : ──>{statement}──┬───>┐
 │      │  └<───── , <──┘                        │    │
 │      └<──────────────────── ; <───────────────┘    │
 │                                                    │
 ├<───────────────────────────────────────────────────┘
 │                      └<─── ; <───┘
 ├───> otherwise ───┬───>{statement}───┬───>┐
 │                  └<─────── ; <──────┘    │
 ├<─────────────────────────────────────────┘
 │
 └──> end ──────────────────────────────────────────────>
```

The case statement provides you with a multiple branch based upon the evaluation of an expression. This statement consists of an expression called the selector and a list of statements. The selector must be of scalar type (except type REAL). Each statement is prefixed with one or more ranges of the same type as the selector; each range is separated by a comma. Each range designates one or more values called case labels.

Pascal/VS evaluates the expression and executes the statement whose case label equals the value of the expression. If no case label equals the value of the expression, then the otherwise statement is executed if it is present; if there is no otherwise statement and the %CHECK CASE option is on, then a runtime error will result. If the checking is not enabled the results will not be predictable.

The range values of a case statement may be written in any order. However, you may not designate the same case label on more than one statement.

Example:

```
type
  SHAPE = (TRIANGLE, RECTANGLE,
           SQUARE,   CIRCLE);
  COORDINATES =
    record
      X,Y    : REAL;
      AREA   : REAL;
      case S : SHAPE of
        TRIANGLE:
           (SIDE : REAL;
            BASE : REAL);
        RECTANGLE:
           (SIDEA,SIDEB : REAL);
        SQUARE:
           (EDGE : REAL);
        CIRCLE:
           (RADIUS : REAL)
    end;
var
  COORD  : COORDINATES;
   ...
with COORD do
  case S of
  TRIANGLE:
    AREA := 0.5 * SIDE * BASE;
  RECTANGLE:
    AREA := SIDEA * SIDEB;
  SQUARE:
    AREA := SQR(EDGE);
  CIRCLE:
    AREA := 3.14159 * SQR(RADIUS)
  end;
```

The Case Statement

Example:

```
 type
   RANK = (ACE,  TWO,   THREE,FOUR,
           FIVE, SIX,   SEVEN,EIGHT,
           NINE, TEN,   JACK, QUEEN,
           KING);
   SUIT = (SPADE,HEART,DIAMOND,CLUB);
   CARD = record
          R    : RANK;
          S    : SUIT
          end;
 var
   POINTS : INTEGER;
   A_CARD : CARD;
      ...
 case A_CARD.R of
   ACE:
     POINTS := 11;
   TWO..TEN:
     POINTS := ORD(A_CARD.R)+1
+  otherwise
+    POINTS := 10
 end;
```

The Case Statement with otherwise

9.4 THE COMPOUND STATEMENT

Syntax:

compound-statement:

```
——> begin ——┬——>{statement}——┬———> end ——————————————————————————————>
            └<——————— ; <—————┘
```

The compound statement serves to bracket a series of statements that are to be executed sequentially. The reserved words "begin" and "end" delimit the statement. Semicolons are used to separate each statement in the list of statements.

Example:

```
if A > B then
  begin       (* swap A and B        *)
    TEMP := A;
    A    := B;
    B    := TEMP
  end
```

Compound Statement

9.5 THE CONTINUE STATEMENT

```
Syntax:

continue-statement:

---> continue ---------------------------------------------------------->
```

The continue statement permits the continuation of an iterative statement (i.e. for, while and repeat). This statement is effectively a goto to the end of the innermost iterative statement. The termination condition is tested and the loop will terminate or will iterate depending on the result of the test.

Example:

```
repeat
  ...
   if A > B then continue;
  ...
(* execution resumes here *)
until P = nil
```

The Continue Statement

9.6 THE EMPTY STATEMENT

Syntax:

empty-statement:

————————————————————————————>

The empty statement is used as a place holder and has no effect on the execution of the program. This statement is often useful when you wish to place a label in the program but do not want it attached to another statement (such as, at the end of a compound statement). The empty statement is also useful to avoid the ambiguity that arises in nested if statements. You may force a single else-clause to be paired with the outer nested if statement (see page 84) by using an empty statement.

```
if b1 then
  if b2 then
    s1
  else
                    (* empty-statement *)
else
  s2
```

9.7 THE FOR STATEMENT

```
Syntax:

for-statement:

---> for --->{id}---> := --->{expr}--+--> to -------+--->{expr}--->+
                                     +--> downto --->+              |
                          +<----------------------------------------+
                          |
                          +-------------> do --->{statement}--------------->
```

The for statement repeatedly executes a statement while the control variable is assigned a series of values. The value of the control variable is incremented (to) or decremented (downto) for each iteration of the loop. The increment (decrement) is computed by the SUCC (PRED) function. That is, the control variable is changed to the suceeding (preceeding) value of the type of the control variable.

The for statement initializes the control variable to the first expression. Prior to each execution of the component statement, the control variable is compared less than or equal to (to), or greater than or equal to (downto) the second expression. Pascal/VS computes the value of the second expression at the beginning of the for statement and uses the result for the duration of the statement. Thus the ending value expression is computed once and can not be changed during the for statement.

The control variable must be an automatic variable which is declared in the immediately enclosing routine. Also, it may not be subscripted, field qualified or referenced through a pointer. The type of the control variable must be a scalar type.

The executed statement must not alter the control variable. If the control variable is altered within the loop, the resultant loop execution is not predictable. The value of the control variable after the for statement is undefined (you should not expect the control variable to contain any particular value).

Given the following statement

```
for I := expr1 to expr2 do stmt
```

where I is an automatic scalar variable; expr1and expr2 are scalar expressions which are type-compatible with I; and 'stmt' is any arbitrary statement. The following compound statement is functionally equivalent:

```
begin
  TEMP1 := expr1;
  TEMP2 := expr2;
  if TEMP1 <= TEMP2 then
    begin
    I := TEMP1;
    repeat
      stmt;
      if I = TEMP2 then
        leave;
      I := SUCC(I)
    until FALSE; (*forever*)
    end
end
```

And given the following statement

```
for I := expr1 downto expr2 do stmt
```

where I is an automatic scalar variable; expr1 and expr2 are scalar expressions which are type-compatible with I; and 'stmt' is any arbitrary statement. The following compound statement is functionally equivalent:

```
begin
  TEMP1 := expr1;
  TEMP2 := expr2;
  if TEMP1 >= TEMP2 then
    begin
    I := TEMP1;
    repeat
      stmt;
      if I = TEMP2 then
        leave;
      I := PRED(I)
    until FALSE; (*forever*)
    end
end
```

where 'TEMP1' and 'TEMP2' are compiler generated temporary variables.

Examples:

```
(* find the maximum INTEGER in      *)
(*  an array of INTEGERs            *)
MAX := A[1];
LARGEST := 1;
for I := 2 to SIZE_OF_A do
 if A[I] <  MAX then
   begin
     LARGEST := I;
     MAX := A[I]
   end


(* matrix multiplication: C<-A*B    *)

for I := 1 to N do
  for J:= 1 to N do
    begin
      X := 0.0;
      for K := 1 to N do
        X := A[I,K] * B[K,J] + X;
      C[I,J] := X
    end


(* sum the hours worked this week  *)

SUM := 0;
for DAY := MON to FRI do
  SUM := SUM + TIMECARD[ DAY ]
```

The For Statement

9.8 THE GOTO STATEMENT

Syntax:

goto-statement:

```
————> goto ——>{label}——————————————————————>
```

The goto statement changes the flow of control within the program.

Examples:

```
goto 10
goto ERROR_EXIT
```

The Goto Statement

The label must be declared within the routine that contains the goto statement.

The following restrictions apply to the use of the goto statement:

- You may not branch into a compound statement from a goto statement which is not contained within the statement.
- You may not branch into the then-clause or the else-clause from a goto statement that is outside the if statement. Further, you may not branch between the then-clause and the else-clause.
- You may not branch into a case-alternative from outside the case statement or between case-alternative statements in the same case statement.
- You may not branch into a for, repeat, or while loop from a goto statement that is not contained within the loop.
- You may not branch into a with statement from a goto statement outside of the with statement.
- A goto statement that specifies a label in a surrounding routine is not permitted. This is a restriction in Pascal/VS which is not in standard Pascal.

The following example illustrates legal and illegal goto statements.

```
procedure GOTO_EXAMPLE;
label
  L1,
  L2,
  JUMPIN,
  JUMPOUT;

begin
  goto JUMPIN;       (* not permitted *)
  ...
  begin
JUMPIN:
    ...
    goto JUMPOUT;      (* permitted     *)
    ...
    goto JUMPIN;       (* permitted     *)
  end;
JUMPOUT:
    ...
  if expr then
    L1:goto L2       (* not permitted *)
  else
    L2:goto L1       (* not permitted *)
```

Goto Target Restrictions

9.9 THE IF STATEMENT

Syntax:

if-statement:

```
 ──> if ──>{expr}──> then ──>{statement}──┬──────────────────────────┬──>
                                          └──> else ──>{statement}──>┘
```

The if statement allows you to specify that one of two statements is to be executed depending on the evaluation of a boolean expression. The if statement is composed of an expression and a then-clause and an optional else-clause. Each clause contains one statement.

The expression must evaluate to a BOOLEAN value. If the result of the expression is TRUE, then the statement in the then-clause is executed. If the expression evaluates to FALSE and there is an else-clause, then the statement in the else-clause is executed; if there is no else-clause, control passes to the next statement.

Example:

```
if A <= B then
  A := (A+1.0)/2.0

if ODD(I) then
  J:=J+1
else
  J:=J div 2 + 1
```

The If Statement

Nesting of an if statement within an if statement could be interpreted with two different meanings if only one statement had an else-clause. The following example illustrates the condition that produces the ambiguity. Pascal/VS always assumes the first interpretation. That is, the else-clauses are paired with the innermost if statement.

The following line could be interpreted two ways.

```
if b1 then if b2 then s1 else s2
```

Interpretation 1 (assumed by Pascal/VS)

```
if b1 then
  begin
    if b2 then
      s1
    else
      s2
  end
```

Interpretation 2 (incorrect interpretation)

```
if b1 then
  begin
    if b2 then
      s1
  end
else
  s2
```

9.10 THE LEAVE STATEMENT

Syntax:
leave-statement:
---> leave --->

The leave statement permits the termination of an iterative statement (i.e. for, while and repeat) without the termination condition occurring. This statement is particularly useful in applications which search and in handling error conditions.

Example:

```
 P:=FIRST;
 while P<>nil do
   if P->.NAME = 'JOE SMITH' then
     leave
   else
     P:=P->.NEXT;
 (* P either points to the desired   *)
 (*  data or is nil                  *)
```

The Leave Statement

9.11 THE PROCEDURE CALL

Syntax:

procedure-call:

```
——>{id:procedure}——┬——————————————————————————————————————┬——————————>
                   └——> ( ——┬——>{expr}——┬——> ) ——>————————┘
                            └<———— , <——┘
```

The procedure statement causes the invocation of a procedure. When a procedure is invoked, the actual parameters are substituted for the corresponding formal parameters. The actual parameters must be conformable to the formal parameters. The rules for expression conformability are given in section 5.2 on page 27.

Parameters which are passed by read/write reference (var) may only be variables, never expressions or constants. Also, fields of a packed record may not be passed by var. Parameters passed by value or read-only reference (const) may be any expression.

A procedure invocation that requires no parameters does not use the list of operands.

Example:

```
TRANSPOSE(AN_ARRAY,
          NUM-OF_ROWS,
          NUM-OF_COLUMNS);

MATRIX_ADD(A_ARRAY,
           B_ARRAY,
           C_ARRAY,
           N,M);

XYZ(I+J, K*L)
```

Procedure Invocations

9.12 THE REPEAT STATEMENT

Syntax:

repeat-statement:

```
──> repeat ──────┬──>{statement}──┬──> until ──>{expr}──────────────>
                 └<────── ; <─────┘
```

The statements contained between the statement delimiters repeat and until are executed until the control expression evaluates to TRUE. The control expression must evaluate to type BOOLEAN. Because the termination test is at the end of the loop, the body of the loop is always executed at least once. The structure of the repeat statement allows it to act like a compound statement in that it encloses a list of statements.

Example:

```
repeat
  K := I mod J;
  I := J;
  J := K
until J = 0
```

The Repeat Statement

+ 9.13 THE RETURN STATEMENT

```
+ Syntax:

+ return-statement:

+ ---> return ------------------------------------------------------------->
```

+ The return statement permits an exit + from a procedure or function. This + statement is effectively a goto to an + imaginary label after the last state- + ment within the routine being executed. + If the %CHECK FUNCTION option is ena- + bled, Pascal/VS will insure that a + function has been assigned a value pri- + or to the return from the function. If + a value has not been assigned, a + runtime error will occur.

9.14 THE WHILE STATEMENT

Syntax:

while-statement:

```
———> while ———>{expr}———> do ———>{statement}———————————————————————————>
```

The while statement allows you to specify a statement that is to be executed while a control expression evaluates to TRUE. The control expression must evaluate to type BOOLEAN. The expression is evaluated prior to each execution of the statement.

Example:

```
(* Compute the decimal size of N    *)
(*   assume N >= 1                  *)
I := 0;
J := 1;
while N > 10 do
  begin
    I := I + 1;
    J := J * 10;
    N := N div 10
  end
(* I is the power of ten of the     *)
(*      original N                  *)
(* J is ten to the I power          *)
(* 1 <= N <= 9                      *)
```

The While Statement

9.15 THE WITH STATEMENT

Syntax:

with-statement:

```
---> with ---+--->{variable}---+---> do --->{statement}----------------------->
             +<------- , <-----+
```

The with statement is used to simplify references to a record variable by eliminating an addressing description on every reference to a field. The with statement makes the fields of a record available as if the fields were variables within the nested statement.

The with statement effectively computes the address of a record variable upon executing the statement. Any modification to a variable which changes the address computation will not be reflected in the pre-computed address during the execution of the with statement. The following example illustrates this point.

```
var A : array[ 1..10 ] of
          record
            FIELD : INTEGER
          end;
       ...
I:=1;
with A[ I ] do
  begin
  K := FIELD;     (*K:=A[1].FIELD*)
  I := 2;
  K := FIELD;     (*K:=A[1].FIELD*)
  end;
```

The Address of A is Computed on Entry to the Statement

The comma notation of a with statement is an abbreviation of nested with statements. The names within a with statement are scoped such that the last with statement will take precedence. A local variable with the same name as a field of a record becomes unavailable in a with statement that specifies the record.

Example:

```
type
  EMPLOYEE =
      record
        NAME     : STRING(20);
        MAN_NO   : 0..999999;
        SALARY   : INTEGER;
        ID_NO    : 0..999999
      end;

var
  FATHER : -> EMPLOYEE;
     ...
with FATHER-> do
  begin
    NAME    := 'SMITH';
    MAN_NO  := 666666;
    SALARY  := WEEKLY_SALARY;
    ID_NO   := MAN_NO
  end
```

is equivalent to:

```
begin
  FATHER->.NAME    := 'SMITH';
  FATHER->.MAN_NO  := 666666;
  FATHER->.SALARY  := WEEKLY_SALARY;
  FATHER->.ID_NO   :=FATHER->.MAN_NO
end
```

Note: The variable FATHER is of type pointer to EMPLOYEE, thus the pointer notation must be used to specify the record pointed to by the pointer.

The With Statement

Example:

```
V : record
      V2 : INTEGER;
      V1 : record A : REAL end;
      A : INTEGER
    end;
A : CHAR;
  ...
with V,V1 do
  begin
    V2  := 1;    (* V.V2   := 1     *)
    A   := 1.0;  (* V.V1.A := 1.0   *)
    V.A := 1     (* V.A    := 1     *)
                 (* CHAR A is not   *)
                 (* available here  *)
  end;
A := 'A';        (* CHAR A is now   *)
                 (* available       *)
```

With Statements Can Hide a Variable

10.0 I/O FACILITIES

Input and output are done using the file data structure. The Pascal/VS Programmer's Guide provides more detail on how to use the I/O facilities in a specific operating system. Pascal/VS provides predefined routines which operate on variables of a file type. The routines are:

- RESET
- REWRITE
- READ
- WRITE
- GET
- PUT
- EOF
- + INTERACTIVE
- + OPEN
- + CLOSE

To facilitate input and output operations that require conversion to and from a character representation, the predefined file type TEXT is provided. The type TEXT is predefined as a file of CHAR. Each GET and PUT transfers one CHAR of information. There are additional predefined routines that may be executed on variables of type TEXT that perform the required conversions.

- READLN
- WRITELN
- EOLN
- PAGE
- + COLS

10.1 RESET PROCEDURE

Open a File for Input

```
definition:

procedure RESET(
          F : filetype);

where:

F is a variable of a file type
```

RESET positions the file pointer to the beginning of the file and prepares the file to be used for input. After you invoke RESET the file pointer is pointing to the first data element of the file. If the file is associated with a terminal, the terminal user would be prompted for data when the RESET is executed. This procedure can be thought of as:

Closing the file (if open).

Rewinding the file.

Opening the file for input.

Getting the first component of the file.

10.2 REWRITE PROCEDURE

Open a File for Output

```
definition:

procedure REWRITE(
          F : filetype);

where:

F is a variable of a file type.
```

REWRITE positions the file pointer to the beginning of the file and prepares the file to be used for output. This procedure can be thought of as:

Closing the file (if open).

Rewinding the file.

Opening the file for output.

+ ## 10.3 INTERACTIVE PROCEDURE

Open a File for Interactive Input

```
definition:

procedure INTERACTIVE(
          F : filetype);

where:

F is a variable of a file type.
```

INTERACTIVE positions the file pointer to the beginning of the file and prepares the file to be used for input. This procedure is similar to RESET except that when a file is opened the file is not positioned at the first data element. If the file is a TEXT file, then an explicit READ is required after a READLN in order to advance the file to the next line of data.

+ 10.4 OPEN PROCEDURE

Open a File

```
definition:

procedure OPEN(
          F : filetype;
          S : STRING );

where:

F is a variable of a file type.
S is an optional parameter that is
  used to supply implementation
  dependent information about the
  file.
```

+ OPEN opens the file for processing. The optional string parameter is expected to contain a file name or other system dependent information. You should refer to the Programmer's Guide for a description of the available options.

+ 10.5 CLOSE PROCEDURE

Close a File

```
definition:

procedure CLOSE(
          F : filetype);

where:

F is a variable of a file type.
```

+ CLOSE closes a file; all processing to the file is completed. You must open the file prior to using it again.

10.6 GET PROCEDURE

Position a File to Next Element

> definition:
>
> procedure GET(F : filetype);
>
> where:
> F is a variable of a file type.

GET positions the file pointer to the next component of the file. For example, if the file is defined as an array of 80 characters, then each GET returns the next 80 character record. A GET invocation on a file of type TEXT returns a single character. If the file associated with the variable is not opened for input, it will be opened implicitly.

10.7 PUT PROCEDURE

Position a File to Next Element

> definition:
>
> procedure PUT(F : filetype);
>
> where:
>
> F is a variable of a file type.

PUT releases the current component of the file variable by effectively writing the component to the associated physical file. For example, if the file is defined as an array of 80 characters, then each PUT transfers an 80 character record. A call to PUT with a file of type TEXT transfers a single character. The file associated with the variable must be open for output.

10.8 EOF FUNCTION

Test File for End Of File

```
definition:

function EOF(F:filetype):BOOLEAN;

function EOF:BOOLEAN;

where:

F is a variable of a file type.
```

EOF is a BOOLEAN valued function which returns TRUE if the end-of-file condition is true for the file. This condition occurs in an input file when an attempt is made to read past the last record element of the file. If the file is open for output, this function always returns TRUE.

If the file variable F is omitted, then the function assumes the predefined file INPUT.

Example:

```
(* The following will read all of   *)
(*   the records from File SYSIN    *)
(*   and write then out to SYSOUT   *)

type FREC =
       record
         A,B:INTEGER
       end;

var
  SYSIN,
  SYSOUT: file of FREC;

begin
  RESET(SYSIN);
  REWRITE(SYSOUT);
  while not EOF(SYSIN) do
    begin
      SYSOUT-> := SYSIN->;
      PUT(SYSOUT);
      GET(SYSIN)
    end;
end;
```

10.9 READ AND READLN (TEXT FILES)

Read Data from TEXT File

```
Definition:

procedure READ(
          f : TEXT;
          v : see below);

procedure READLN(
          f : TEXT;
          v : see below);

where:
f is an optional text file
  that is to be used for input.
v is a variable of one of the
  following types:
  - INTEGER (or subrange)
  - CHAR    (or subrange)
  - REAL
  - STRING
  - packed array of CHAR
```

The READ procedure reads character data from the TEXT file f. READ converts character data to conform to the type of the operand. The file parameter is optional; the default file is INPUT.

READLN positions the file at the beginning of the next line. You may use more than one variable on each call by separating each with a comma. The effect is the same as multiple calls to READ.

```
           READ(f,v1,v2)

is equivalent to:

           begin
             READ(f,v1);
             READ(f,v2)
           end

     and

           READLN(f,v1,v2,v3)

is equivalent to:

           begin
             READ(f,v1);
             READ(f,v2);
             READ(f,v3);
             READLN(f);
           end
```

Multiple Variables on READ or READLN

Reading INTEGER Data

INTEGER data from a TEXT file is read by scanning off leading blanks, accepting an optional sign and converting all characters up to the first non-numeric character.

Reading CHAR Data

A variable of type CHAR is assigned the next character in the file.

Reading STRING Data

Characters are read into a STRING variable until the variable has reached its maximum length or until the end of the line is reached.

Reading REAL Data

REAL data is read by scanning off leading blanks, accepting an optional sign and converting all characters up to the first non-numeric character not conforming to the syntax of a REAL number.

Reading packed array of CHAR Data

If the variable is declared as a 'packed array[1..n] of CHAR', characters are stored into each element of the array. This is equivalent to a loop ranging from the lower bound of the array to the upper bound, performing a read operation for each element. If the end-of-line condition should become true before the variable is filled, the rest of the variable is filled with blanks.

Consult the Programmer's Guide for more details on the use of READ and READLN.

```
var
  I,J: INTEGER;
  S: STRING(100);
  CH: CHAR;
  CC: packed array[1..10] of CHAR;
  F: TEXT;
   .
   .
  READLN(F,I,J,CH,CC,S);
```

assume the data is:

36 24 ABCDEFGHIHKLMNOPQRSTUVWXYZ

the variables would be assigned:

I	36
J	24
CH	' '
CC	'ABCDEFGHIJ'
S	'KLMNOPQRSTUVWXYZ'
LENGTH(S)	16

The READ Procedure

10.10 READ (NON-TEXT FILES)

Read Data from Non-TEXT Files

```
Definition:

procedure READ(
            f : file of t;
            v : t);

where:
f is an arbitrary file variable.
v is a variable whose type matches
  the file component type of f
```

Each call to READ will read one file element from file 'f' and assign it to variable 'v'. If the file is not open, the READ procedure will open it prior to assigning to the argument.

READ(f,v) is functionally equivalent to the following compound statement:

```
begin
  v := f->;
  GET(f)
end
```

For more details consult the Programmer's Guide.

10.11 WRITE AND WRITELN (TEXT FILES

Write Data to FIle

```
Definition:

procedure WRITE(
            f : TEXT;
            e : see below);

procedure WRITELN(
            f : TEXT;
            e : see below);

where:
f is an optional TEXT file
  variable.
e is an expression of one of the
  following types:
  - INTEGER (or subrange)
  - CHAR    (or subrange)
  - REAL
  - BOOLEAN
  - STRING
  - packed array[1..n] of CHAR
Pascal/VS accepts a special para-
meter format which is only
allowed in the WRITE routine
for TEXT files.
See the following description.
```

The WRITE procedure writes character data to the TEXT file specified by f. The data is obtained by converting the expression e into an external form. The file parameter is optional; if not specified, the default file OUTPUT is used.

WRITELN positions the file to the beginning of the next line. WRITELN is only applicable to TEXT files. You may use more than one expression on each call by separating each with a comma. The effect is the same as multiple calls to WRITE.

```
          WRITE(f,e1,e2)

is equivalent to:

          begin
            WRITE(f,e1);
            WRITE(f,e2)
          end

and

          WRITELN(f,e1,e2,e3)

is equivalent to:

          begin
            WRITE(f,e1);
            WRITE(f,e2);
            WRITE(f,e3);
            WRITELN(f);
          end
```

Multiple Expressions on WRITE

Pascal/VS supports a specialized form for specifying actual parameters on WRITE and WRITELN to TEXT files. This provides a means by which you can specify the length of the resulting output. Each expression in the WRITE procedure call may be represented in one of three forms:

1. e
2. e : len1
3. e : len1 : len2

The expression e may be of any of the types outlined above and represents the data to be placed on the file. The data is converted to a character representation from the internal form. The expressions len1 and len2 must evaluate to an INTEGER value.

The expression len1 supplies the length of the field into which the data is written. The data is placed in the field justified to the right edge of the field. If len1 specifies a negative value, the data is justified to the left within a field whose length is ABS(len1). If len1 has the value zero, the data will be placed in a field with no padding or truncation.

The len2 expression (form 3) may be specified only if e is an expression of type REAL.

If len1 is unspecified (form 1) then a default value is used according to the table below.

type of expression e	default value of len1
INTEGER	12
REAL	20 (E notation)
CHAR	1
BOOLEAN	10
STRING	LENGTH(expression)
array of CHAR	number elements in the array

Default Field Width on WRITE

<u>Writing INTEGER Data</u>

The expression len1 represents the width of the field in which the integer is to be placed. The value is converted to character format and placed in a field of the specified length. If the field is shorter than the size required to represent the value, digits are truncated from the left (most significant position).

Examples:

```
   Call:                  Result:

 WRITE(1234:6)             '  1234'

 WRITE(1234:-6)            '1234  '

 WRITE(1234:0)             '1234'

 WRITE(1234)               '        1234'

 WRITE(1234:3)             '234'
```

<u>Writing CHAR Data</u>

The value of len1 is used to indicate the width of the field in which the character is to be placed. If len1 is not specified, a field width of 1 is assumed. If len1 is greater than 1 then the character will be padded on the left with blanks; if len1 is negative, then the character will be padded on the right.

Example:

```
   call:           Result:

 WRITE('a':6)            '     a'
 WRITE('a':-6)           'a     '
```

<u>Writing REAL Data</u>

REAL expressions may be printed with any one of the three operand formats. If len1 is not specified (form 1), the result will be in scientific notation in a 20 character field.

If len1 is specified and len2 is not (form 2), the result will be in scientific notation but the number of characters in the field will be the value of len1.

If both len1 and len2 are specified (form 3), the data will be written in fixed point notation in a field with length len1; len2 specifies the number of digits that will appear to the right of the decimal point. The REAL expression is always rounded to the last digit to be printed.

Examples:

Call:	Result:
WRITE(3.14159:10)	' 3.142E+00'
WRITE(3.14159)	' 3.1415900000000E+00'
WRITE(3.14159:10:4)	' 3.1416'

Writing BOOLEAN Data

The expression len1 is used to indicate the width of the field in which the
+ boolean is to be placed. If the width
+ is less than 6, then either a 'T' or
+ 'F' will be printed. Otherwise, 'TRUE'
+ or 'FALSE' will be sent to the file.
+ The data is placed in the field and
+ justified according to the previously
+ stated rules.

Examples:

	Call:	Result:
	WRITE(TRUE:10)	' TRUE'
+	WRITE(TRUE:-10)	'TRUE '
+	WRITE(FALSE:2)	' F'
	WRITE(TRUE:0)	'TRUE'

Writing STRING Data

The second expression is used to indicate the width of the field in which the string is to be placed. The data is placed in the field and justified according to the previously stated rules.

Examples:

	Call:	Result:
	WRITE('abcd':6)	' abcd'
+	WRITE('abcd':-6)	'abcd '
+	WRITE('abcd':2)	'ab'
+	WRITE('abcd':0)	'abcd'

Writing Packed Array of CHAR Data

The second expression is used to indicate the width of the field in which the array is to be placed. The data is placed in the field and justified according to the previously stated rules.

Examples:

```
var
  A : packed
      array[ 1..4] of CHAR;
  .
  .
  A := 'abcd';
  .
  .
```

	Call:	Result:
	WRITE(A:6)	' abcd'
+	WRITE(A:-6)	'abcd '
+	WRITE(A:2)	'ab'
+	WRITE(A:0)	'abcd'

10.12 WRITE (NON-TEXT FILES)

Write Data to Non-TEXT Files

Definition:

```
procedure WRITE(
          f : file of t;
          e : t);
```

where:
f is an arbitrary file variable.
e is an expression whose type matches the file component type of f

Each call to WRITE will write the value of expression e to file 'f'.

WRITE(f,e) is functionally equivalent to the following compound statement:

```
begin
```

```
  f-> := e;
  PUT(f)
 end
```

For more details consult the Programmer's Guide.

10.13 EOLN FUNCTION

Test a File for End of Line

<u>Definition</u>: function EOLN(f: TEXT):BOOLEAN; function EOLN:BOOLEAN; <u>where</u>: f is a TEXT file set to input.

The EOLN function returns a BOOLEAN result of TRUE if TEXT file f is positioned to an end-of-line character; otherwise, it returns FALSE.

If EOLN(f) is true, then f-> has the value of a blank. That is, when EOLN is TRUE the file is positioned to a blank. This character is not in the file but will appear as if it were. In many applications the extra blank will not affect the result; in those instances where the physical layout of the data is significant you must be sensitive to the EOLN condition.

If the file variable F is omitted, then the function assumes the predefined file INPUT.

10.14 PAGE PROCEDURE

Force Skip to Next Page

```
Definition:

procedure PAGE( var f: TEXT );

where:

f is a TEXT file set to
  output.
```

This procedure causes a skip to the top of the next page when the text-file is printed. The file parameter is optional and defaults to the standard file variable OUTPUT.

+ 10.15 COLS FUNCTION

Determine Current Column

```
Definition:

function COLS(
   var f: TEXT ) : INTEGER;

where:

f is a TEXT file set to
  output.
```

+ This function returns the current column number (position of the next character to be written) on the output file designated by the file variable. You may force the output to a specific column with the following code:

```
+   if TAB > COLS(F) then
+     WRITE(F,' ':TAB-COLS(F));
```

+ The file name is never defaulted on the COLS procedure.

11.0 EXECUTION LIBRARY FACILITIES

The runtime library consists of those routines that are predefined in Pascal/VS. In addition to the routines described in this chapter, Pascal/VS provides routines with which to do input and output. Consult the I/O chapter for a description of those routines. The predefined procedures and functions are:

+ • MARK Procedure
+ • RELEASE Procedure
• NEW Procedure
• DISPOSE Procedure
• PACK Procedure
• UNPACK Procedure
+ • LBOUND Function
+ • HBOUND Function
+ • MIN Function
+ • MAX Function
+ • LOWEST Function
+ • HIGHEST Function
+ • SIZEOF Function
• ORD Function
• CHR Function
+ • Scalar Conversion
• PRED Function
• SUCC Function
• ODD Function
• ABS Function
+ • FLOAT Function
• TRUNC Function
• ROUND Function
+ • STR Function
+ • LENGTH Function
• SIN Function
• COS Function
• ARCTAN Function
• EXP Function
• LN Function
• SQRT Function
• SQR Function
+ • RANDOM Function
+ • SUBSTR Function
+ • TRIM Function
+ • LTRIM Function
+ • DELETE Function
+ • COMPRESS Function
+ • INDEX Function
+ • TOKEN Function
+ • TRACE Procedure
+ • HALT Procedure
+ • DATETIME Procedure
+ • CLOCK Function
+ • PARMS Function
+ • RETCODE Procedure

11.1 MEMORY MANAGEMENT ROUTINES

These routines provide means by which you can control the allocation of dynamic variables.

11.1.1 MARK PROCEDURE

Mark Heap

```
definition:

procedure MARK(
  var     P          : pointer );

where:

P is a pointer to any type
```

The MARK procedure allocates a new heap. All dynamic variables are allocated from an area of storage called the heap. The predefined procedure RELEASE frees a heap created by MARK. Thus, heaps are created and destroyed in a stack fashion. The predefined procedure NEW allocates a dynamic variable from the most recent heap. The predefined procedure DISPOSE de-allocates a dynamic variable from the heap.

11.1.2 RELEASE PROCEDURE

Release Heap

```
definition:

procedure RELEASE(
  var     P          : pointer );

where:

P is a pointer to any type.
```

RELEASE returns all heap storage that was allocated since the matching MARK. The parameter of RELEASE is the same one as was specified on a previous call to MARK; it is through this parameter that the matching MARK is identified. RELEASE permits dynamic variables to be returned in blocks. RELEASE sets the pointer to nil.

```
type
  MARKP = ->INTEGER;
  LINKP = ->LINK;
  LINK = record
           NAME: STRING(30);
           NEXT: LINKP
         end;
var
  P     : MARKP;
  Q1,
  Q2,
  Q3    : LINKP;
begin
  ...
  MARK(P);
  ...
  NEW(Q1);
  NEW(Q2);
  NEW(Q3);
  ...
  (* Frees Q1, Q2 and Q3        *)
  RELEASE(P);
  ...
end;
```

Example of MARK and RELEASE

11.1.3 NEW PROCEDURE

Allocate Dynamic Variable

```
definition:

form 1:
procedure NEW(
  var      P          : pointer );

form 2:
procedure NEW(
  var      P1         : pointer;
           t1,t2... : scalar);

where:

P   is a pointer to any type
    except a dynamic array.
P1  is a pointer to a record
    type with variants
P2  is a pointer to a dynamic
    array.
t1,t2... are scalar constants
    representing tag fields
```

```
type
  LINKP = ->LINK;
  LINK = record
           NAME: STRING(30);
           NEXT: LINKP
         end;
var
  P,
  HEAD : LINKP;
  ...
begin
  ...
  NEW(P);
  with P-> do
    begin
      NAME := '';
      NEXT := HEAD;
    end;
  HEAD := P;
  ...
end;
```

Example of using Simple Form of Procedure NEW

The NEW procedure allocates a dynamic variable from the most recent heap and sets the pointer to point to the variable.

form 1

The first form of procedure NEW allocates the amount of storage that is necessary to represent a value of the type to which the pointer refers. If the type of the dynamic variable is a record with a variant part, the space allocated is the amount required for the record when the largest variant is active.

form 2

The second form is used to allocate a variant record when it is known which variant (and sub-variants) will be active, in which case the amount of storage allocated will be no larger than necessary to contain the variant specified. The scalar constants are tag field values. The first one indicates a particular variant in the record which will be active; subsequent tags indicate active sub-variants, sub-sub-variants, and so on.

Note: This procedure does not set tag fields. The tag list only serves to indicate the amount of storage required; it is the programmer's responsibility to set the tag fields after the record is allocated.

```
type
  AGE = 0..100;
  RECP = ->REC;
  REC  =
     record
       NAME: STRING(30);
       case HOW_OLD: AGE of
        0..18:
          (FATHER: RECP);
        19..100:
          (case MARRIED: BOOLEAN of
            TRUE: (SPOUSE: RECP);
            FALSE: ()
          )
     end;
var
  P     : RECP;
  ...
begin
  ...
  NEW(P,18);
  with P-> do begin
    NAME := 'J. B. SMITH, JR'
    HOW_OLD := 18;
    NEW(FATHER,54,TRUE);
    with FATHER-> do begin
      NAME := 'J. B. SMITH';
      HOW_OLD := 54;
      MARRIED := TRUE;
      NEW(SPOUSE,50,TRUE);
        ...
    end (*with father->*);
  end (*with p->*);
  ...
end;
```

Using NEW for Allocating Records with Variants

11.1.4 DISPOSE PROCEDURE

De-allocate Dynamic Variable

<u>definition</u>:

```
procedure DISPOSE(
  var     P         : pointer);
```

<u>where</u>:

P is any pointer type.

DISPOSE returns storage for a dynamic variable. You may de-allocate a dynamic variable from any heap. This procedure only returns the storage referred to by the pointer and does not return any storage which the dynamic variable references. That is, if the dynamic variable is part of a linked list, you must explicitly DISPOSE of every element of the list. DISPOSE sets the pointer to nil. If you have other pointers which reference the same DISPOSEd dynamic variable, then it is your responsibility not to use these pointers because the dynamic variable which they represented is no longer allocated.

11.2 DATA MOVEMENT ROUTINES

These routines provide you with convenient ways to handle large amounts of data movement efficiently.

11.2.1 PACK PROCEDURE

Copy Unpacked Array to Packed Array

```
definition:

procedure PACK(
  const   SOURCE : array-type;
          INDEX  : index_of_source;
  var     TARGET : pack_array_type)

where:

SOURCE is an array.
INDEX  is an expression which is
       compatible with the index
       of SOURCE.
TARGET is a variable of type packed
       array.
```

This procedure fills the target array with elements from the source array starting with the index I where the target array is packed. The types of the elements of the two arrays must be identical. This procedure operates as:

```
Given:
  A : array[m..n] of T;
  Z : packed array[u..v] of T;

Call:
  PACK(A, I, Z);

Operation:
  k := I;
  for j := LBOUND(Z) to HBOUND(Z) do
    begin
    Z[j] := A[k];
    k := SUCC(k)
    end;

Where:
  j and k are temporary variables.
```

It is an error if the number of elements in Z is greater than the number of elements in A starting with the Ith element to the end of the array.

11.2.2 UNPACK PROCEDURE

Copy Packed Array to Unpacked Array

```
definition:

procedure UNPACK(
  var     SOURCE : pack_array_type;
  const   TARGET : array-type;
          INDEX  : index_of_target)

where:

SOURCE is a packed array.
TARGET is a variable of type array.
INDEX  is an expression which is
       compatible with the index
       of TARGET.
```

This procedure fills the target array with elements from the source array where the source array is packed. The type of the elements of the two arrays must be identical. This procedure operates as:

```
Given:
  A : array[m..n] of T;
  Z : packed array[u..v] of T;

Call:
  UNPACK(Z, A, I);

Operation:
  k := I;
  for j := LBOUND(Z) to HBOUND(Z) do
    begin
    A[k] := Z[j];
    k := SUCC(k)
    end;

Where:
  j and k are temporary variables.
```

It is an error if the number of elements in Z is greater than the number of elements in A starting with the Ith element to the end of the array.

11.3 DATA ACCESS ROUTINES

These routines provide you a means to inquire about compile and run time bounds and values.

11.3.1 LBOUND FUNCTION

Lower Bound of Array

```
definition:

function LBOUND(
          V     : arraytype;
          I     : integer-const)
                : scalar;

function LBOUND(
          T     : type-identifier;
          I     : integer-const)
                : scalar;

where:

V is a variable which is declared
  as an array type.
T is an type identifier declared
  as an array.
I is an positive integer valued
  constant expression and is
  optional.
```

The LBOUND function returns the lower bound of an index to an array. The array may be specified in two ways:

- an identifier which was declared as an array type via the type construct;
- a variable which is of an array type.

The value returned is of the same type as the type of the index. The second parameter defines the dimension of the array for which the lower bound is returned. It is assumed to be "1" if it is not specified.

Example:

```
type
  GRID = array[-10..10,-10..10] of
             REAL;
var
  A       : array[ 1..100 ] of ALFA;
  B       : array[ 1..100 ] of
             of array[ 0..9 ] of CHAR;
  .
  .
  LBOUND( A )          is 1
  LBOUND( GRID, 1)     is -10
  LBOUND( B, 2 )       is 0
  LBOUND( B[1] )       is 0
```

The LBOUND Function

+ 11.3.2 HBOUND FUNCTION

Upper Bound of Array

```
definition:

function HBOUND(
           V     : arraytype;
           I     : integer-const)
                 : scalar;

function HBOUND(
           T     : type-identifier;
           I     : integer-const)
                 : scalar;

where:

V is a variable which is declared
  as an array type.
T is an type identifier declared
  as an array.
I is an positive integer-valued
  constant expression and is
  optional.
```

The HBOUND function returns the upper bound of an index to an array. The array may be specified in two ways:

- an identifier which was declared as an array type via the type construct;
- a variable which is of an array type.

The value returned is of the same type as the type of the index. The second parameter defines the dimension of the array for which the upper bound is returned. It is assumed to be "1" if it is not specified.

Example:

```
type
  GRID = array[-10..10,-10..10] of
             REAL;

var
  A      : GRID;
  B      : array[ 1..100 ] of
             of array[ 0..9 ] of CHAR;
  .
  .
  HBOUND( A )        is 10
  HBOUND( GRID )     is 10
  HBOUND( B, 2 )     is 9
  HBOUND( B[1] )     is 9
```

The HBOUND Function

+ 11.3.3 LOWEST FUNCTION

Lowest Value of a Scalar

```
definition:

function LOWEST(
           S         : scalar-type)
                     : scalar;

where:

S is an identifier that has been
  declared as a scalar type, or
  a variable which is of a scalar
  type.
```

This function returns the lowest value that is in the scalar type. The operand may be either a type identifier or a variable. If the operand is a type identifier, the value of the function is the lowest value that a variable of that type may be assigned. If the operand is a variable, the value of the function is the lowest value that the variable may be assigned.

If the argument S refers to a record-type which has a variant part, and if no tag values are specified, then the storage required for the record with the largest variant will be returned.

Example:

```
type
  DAYS    = (SUN,  MON,  TUES,  WED,
             THU,  FRI,  SAT);
  SMALL   = 0 .. 31;
var
  I       : INTEGER;
  J       : 0 .. 255;
  .
  .
  LOWEST(DAYS)     is SUN
  LOWEST(BOOLEAN)  is FALSE
  LOWEST(SMALL)    is 0
  LOWEST(I)        is MININT
  LOWEST(J)        is 0
```

The LOWEST Function

+ 11.3.4 HIGHEST FUNCTION

Highest Value of a Scalar

```
definition:

function HIGHEST(
         S        : scalar-type)
                  : scalar;

where:

S is an identifier that has been
  declared as a scalar type, or
  a variable which is of a scalar
  type.
```

+ This function returns the highest value that is in the scalar type. The operand may be either a type identifier or a variable. If the operand is a type identifier, the value of the function is the highest value that a variable of that type may be assigned. If the operand is a variable, the value of the function is the highest value that the variable may be assigned.

+ Example:

```
type
  DAYS    = (SUN,  MON,  TUES,  WED,
             THU,  FRI,  SAT);
  SMALL   = 0 .. 31;
var
  I       : INTEGER;
  J       : 0 .. 255;
  .
  .
  HIGHEST(DAYS)     is SAT
  HIGHEST(BOOLEAN)  is TRUE
  HIGHEST(SMALL)    is 31
  HIGHEST(I)        is MAXINT
  HIGHEST(J)        is 255
```

The HIGHEST Function

+ 11.3.5 SIZEOF FUNCTION

Allocation Size of Data

```
definition:

function SIZEOF(
         S        : anytype)
                  : INTEGER;

function SIZEOF(
         S        : recordtype;
     t1,t2,...    : tags);
                  : INTEGER;

where:

S is an identifier that has been
  declared as a type, or any
  variable.
```

+ The SIZEOF function returns the amount of storage in bytes required to contain the variable or a variable of the type specified.

+ If S is a record variable or a type identifier of a record, it may be followed by tag list which defines a particular variant configuration of the record. In this case the function will return the amount of storage required within the record to contain that variant configuration.

11.4 CONVERSION ROUTINES

These routines allow for conversions between one data type and another. Other type conversions must be programmed.

11.4.1 ORD FUNCTION

Ordinal Value of Scalar

```
definition:

function ORD(
            S          : scalar )
                       : INTEGER;

where:

S is may be any scalar type or
  a pointer.
```

This function returns an integer that corresponds to the ordinal value of the scalar. If the operand is of type CHAR then the value returned is the position in the EBCDIC character set for the character operand. If the operand is an enumerated scalar, then it returns the position in the enumeration (beginning at zero); for example, if COLOR = (RED, YELLOW, BLUE), then ORD(RED) is 0 and ORD(BLUE) is 2.

If the operand is a pointer, then the function returns the machine address of the dynamic variable referenced by the pointer. Although pointers can be converted to INTEGERs, there is no function provided to convert an INTEGER to a pointer.

11.4.2 CHR FUNCTION

Integer to Character Conversion

```
definition:

function CHR(
            I          : INTEGER )
                       : CHAR;

where:

I is an INTEGER expression that is
  to be interpreted as a character.
```

This function is the inverse function to ORD for characters. That is, 'ORD(CHR(I))=I' if I is in the subrange:

ORD(LOWEST(CHAR))..ORD(HIGHEST(CHAR))

If the operand is not within this range and checking is enabled then a runtime error will result, otherwise the result is unpredictable.

+ 11.4.3 SCALAR CONVERSION

Integer to Scalar Conversion

```
definition:

function type-id(
            I          : INTEGER)
                       : scalar-type;

where:

I  is an integer valued expression
   that is to be converted to an
   enumerated scalar.
```

+ Every type identifier for an enumerated scalar or subrange scalar can be used as a function that converts an integer into a value of the enumerated scalar. These functions are the inverse of ORD.

Example:

```
type
  DAYS    = (SUN,  MON,  TUES,  WED,
             THU,  FRI,  SAT);
  .
  .
  DAYS(0)           is SUN
  DAYS(3)           is WED
  DAYS(6)           is SAT
  DAYS(7)           is an error
  BOOLEAN(0)        is FALSE
  BOOLEAN(1)        is TRUE
```

The Enumerated Scalar Function

+ 11.4.4 FLOAT FUNCTION

Integer to Real Conversion

```
definition:

function FLOAT(
          I          : INTEGER )
                     : REAL;

where:

I is an INTEGER valued expression.
```

+ This function converts an INTEGER to a REAL. Pascal/VS will convert an INTEGER to a REAL implicitly if one operand of an arithmetic or relation operator is REAL and the other is INTEGER. This function is useful in making the conversion explicit in the program.

11.4.5 TRUNC FUNCTION

Real to Integer Conversion

```
definition:

function TRUNC(
           R          : REAL )
                      : INTEGER;

where:

R is a REAL valued expression.
```

This function converts a REAL expression to an INTEGER by truncating the operand toward zero.

Examples:

```
TRUNC( 1.0) is  1
TRUNC( 1.1) is  1
TRUNC( 1.9) is  1
TRUNC( 0.0) is  0
TRUNC(-1.0) is -1
TRUNC(-1.1) is -1
TRUNC(-1.9) is -1
```

11.4.6 ROUND FUNCTION

Real to Integer Conversion

```
definition:

function ROUND(
           R          : REAL )
                      : INTEGER;

where:

R is a REAL valued expression.
```

This function converts a REAL expression to an INTEGER by rounding the operand. This function equivalent to

```
if R > 0.0 then
  ROUND := TRUNC(R + 0.5)
else
  ROUND := TRUNC(R - 0.5)
```

Examples:

```
ROUND( 1.0) is  1
ROUND( 1.1) is  1
ROUND( 1.9) is  2
ROUND( 0.0) is  0
ROUND(-1.0) is -1
ROUND(-1.1) is -1
ROUND(-1.9) is -2
```

+ ## 11.4.7 STR FUNCTION

+ Convert to String

```
definition:

function  STR(
          X         : CHAR or packed
                      array[1..n] of
                        CHAR )
                    : STRING;

where:

X is CHAR or packed array[1..n] of
  CHAR expression.
```

+ This function converts either a CHAR or + packed array[1..n] of CHAR to a STRING. + Pascal/VS will implicitly convert a + STRING to a CHAR or packed array[1..n] + of CHAR on assignment, but all other + conversions require you to explicitly + state the conversion. You may assign a + CHAR to an packed array[1..n] of CHAR + by either:

```
var
   AOC : ALPHA;
   CH  : CHAR;
   ...
   AOC := STR(CH);
         or
   AOC := ' '; AOC[1] := CH;
```

11.5 MATHEMATICAL ROUTINES

These routines defined various mathematical transformations.

+ 11.5.1 MIN FUNCTION

MINimum Value of Scalars

```
definition:

function MIN(
          E0,
          .
          .
          En          : scalar-type)
                      : scalar-type;

where:

Ei is an expression of a scalar
   type. All parameters must be
   of the same type except where
   noted below.
```

+ The MIN function returns the minimum value of two or more expressions. The parameters may be of any scalar type, including REAL. The parameters may be a mixture of INTEGER and REAL expressions, in which case, the result will be of type REAL. In all other cases, the parameters must be conformable to each other.

+ 11.5.2 MAX FUNCTION

Maximum Value of Scalars

```
definition:

function MAX(
          E0,
          .
          .
          En          : scalar-type)
                      : scalar-type;

where:

Ei is an expression of a scalar
   type. All parameters must be
   of the same type except where
   noted below.
```

+ The MAX function returns the maximum value of two or more parameters. The parameters may be of any scalar type, including REAL. They may be a mixture of INTEGER and REAL expressions, in which case, the result will be of type REAL. In all other cases, the parameters must be conformable to each other.

11.5.3 PRED FUNCTION

Predecessor Value of a Scalar

```
definition:

function PRED(
            S          : scalar)
                       : scalar;

where:

S is any scalar expression.
```

This function returns the predecessor value of the parameter expression. The PRED of the first element of an enumerated scalar is an error. If the option %CHECK is ON, a runtime error will be raised if the PRED of the first element is attempted. If the checking is not performed, the results of the PRED of the first value is not defined. PRED(TRUE) is FALSE and PRED('B') is 'A'. The PRED of an INTEGER is equivalent to subtracting one. PRED of a REAL argument is an error.

11.5.4 SUCC FUNCTION

Successor Value of a Scalar

```
definition:

function SUCC(
            S          : scalar)
                       : scalar;

where:

S is any scalar expression.
```

This function returns the successor value of the parameter expression. The SUCC of the last element of an enumerated scalar is an error. If the option %CHECK is ON, a runtime error will be raised if the SUCC of the last element is attempted. If the checking is not performed, the results of the SUCC of the last value is not defined. SUCC(FALSE) is TRUE and SUCC('B') is 'C'. The SUCC of an INTEGER is equivalent to adding one. SUCC of a REAL argument is an error.

11.5.5 ODD FUNCTION

Test for Integer is Odd

```
definition:

function ODD(
           I          : INTEGER)
                      : BOOLEAN;

where:

I is an INTEGER to be tested
  for being odd.
```

This function returns TRUE if the parameter I is odd, or FALSE if it is even.

11.5.6 ABS FUNCTION

Absolute Value

```
definition:

function ABS(
           I          : INTEGER )
                      : INTEGER;

function ABS(
           R          : REAL)
                      : REAL;

where:

I is an INTEGER expression.
R is a REAL expression.
```

The ABS function returns either a REAL value or an INTEGER value depending the type of its parameter. The result is the absolute value of the parameter.

11.5.7 SIN FUNCTION

Compute Sine

```
definition:

function SIN(
          X          : REAL)
                     : REAL;

where:

X is an expression that evaluates
  to a REAL value.
```

The SIN function computes the sine of parameter X, where X is expressed in radians.

11.5.8 COS FUNCTION

Compute Cosine

```
definition:

function COS(
          X          : REAL)
                     : REAL;

where:

X is an expression that evaluates
  to a REAL value.
```

The COS function computes the cosine of the parameter X, where X is expressed in radians.

11.5.9 ARCTAN FUNCTION

Compute Arctangent

```
definition:

function ARCTAN(
           X          : REAL)
                      : REAL;

where:

X is an expression that evaluates
  to a REAL value.
```

The ARCTAN function computes the arctangent of parameter X. The result is expressed in radians.

11.5.10 EXP FUNCTION

Compute Exponential

```
definition:

function EXP(
           X          : REAL)
                      : REAL;

where:

X is an expression that evaluates
  to a REAL value.
```

The EXP function computes the value of the base of the natural logarithms, e, raised to the power expressed by parameter X.

11.5.11 LN FUNCTION

Compute Natural Log

```
definition:
function LN(
          X          : REAL)
                     : REAL;

where:

X is an expression that evaluates
  to a REAL value.
```

The LN function computes the natural logarithm of the parameter X.

11.5.12 SQRT FUNCTION

Compute Square Root

```
definition:

function SQRT(
          X          : REAL)
                     : REAL;

where:

X is an expression that evaluates
  to a REAL value.
```

The SQRT function computes the square root of the parameter X. If the argument is less than zero, a run time error is produced.

11.5.13 SQR FUNCTION

Compute Square

<u>definition</u>:

```
function SQR(
           X        : REAL)
                    : REAL

function SQR(
           X        : INTEGER)
                    : INTEGER
```

<u>where</u>:

X is an expression that evaluates to a REAL or INTEGER value.

The SQR function computes the square of the argument. If the argument is of type REAL, then a REAL result is returned, otherwise the function returns an INTEGER.

11.5.14 RANDOM FUNCTION

Compute a Random Number

<u>definition</u>:

```
function RANDOM(
           S        : INTEGER);
                    : REAL;
```

<u>where</u>:

S is an expression that evaluates to an INTEGER value.

The RANDOM function returns a pseudo random number in the range >0.0 and <1.0. The parameter S is called the seed of the random number and is used to specify the beginning of the sequence. RANDOM always returns the same value when called with the same non zero seed. If you pass a seed value of 0, RANDOM will return the next number as generated from the previous seed. Thus, the general way to use this function is to pass it a non zero seed on the first invocation and a zero value thereafter.

It is suggested that you chose an inital value for the seed which is odd.

11.6 STRING ROUTINES

These routines provide convenient means of operating on string data.

+ 11.6.1 LENGTH FUNCTION

Length of String

```
definition:

function LENGTH(
          S         : STRING)
                    : 0..255;

where:

S is a STRING valued expression.
```

+ This function returns the current length of the parameter. The value will be in the range 0..255.

+ 11.6.2 SUBSTR FUNCTION

Obtain Substring

```
definition:

function SUBSTR(
  const    SOURCE   : STRING;
           START    : INTEGER;
           LEN      : INTEGER)
                    : STRING;

where:
SOURCE is a STRING expression from
    which a portion will be
    returned.
START is an INTEGER expression that
    designates the first position
    in the SOURCE to be returned.
LEN is an INTEGER expression that
    defines the number of
    characters to be returned.
```

+ The SUBSTR function returns a specified portion of the first parameter. The source is indexed from 1 to the LENGTH of SOURCE. The function treats the source as if it were surrounded by blanks on either side, thus if the LEN exceeds the LENGTH of the source, the returned value will be padded on the right with blanks. Similarly, if the START expression is less than one, blanks will be supplied on the left.

+ Examples:

```
var
  S   : STRING;
  ...
  S := 'ABCDE';
  ...
  SUBSTR('ABCDE',2,3) yields 'BCD'
  SUBSTR(S,2,5) yields 'BCDE '
  SUBSTR(S,0,8) yields ' ABCDE  '
```

+ 11.6.3 DELETE FUNCTION

+ Delete Substring

```
+ definition:

+ function DELETE(
+   const    SOURCE    : STRING;
+            START     : INTEGER;
+            LEN       : INTEGER)
+                      : STRING;

+ where:

+ SOURCE is a STRING expression from
+     which a portion will be
+     returned.
+ START is an INTEGER expression that
+     designates the first position
+     in the SOURCE to be returned.
+ LEN is an INTEGER expression that
+     defines the number of
+     characters to be returned.
```

+ The DELETE function removes a specified
+ portion of the value of the first
+ parameter and returns the result. The
+ source is indexed from 1 to the LENGTH
+ of the source. An attempt to delete a
+ portion of the source beyond the length
+ is ignored.

+ Examples:

```
+  var
+    S   : STRING;
+    ...
+    S := 'ABCDE';
+    ...
+    DELETE('ABCDE',2,3) yields 'AE'
+    DELETE(S,5,3) yields 'ABCD'
+    DELETE(S,0,3) yields 'CDE'
```

+ 11.6.4 TRIM FUNCTION

+ Remove Trailing Blanks

```
+ definition:

+ function TRIM(
+   const    SOURCE    : STRING)
+                      : STRING;

+ where:

+ SOURCE is the STRING to be trimmed.
```

+ The TRIM function returns the parameter
+ value with all trailing blanks removed.

+ Example:

```
+  TRIM(' A B   ') yields ' A B'
+  TRIM('       ') yields ''
```

+ 11.6.5 LTRIM FUNCTION

+ Remove Leading Blanks

```
definition:

function LTRIM(
  const   SOURCE   : STRING)
                   : STRING;

where:

SOURCE is the STRING to be trimmed.
```

+ The LTRIM function returns the parameter value with all leading blanks removed.

+ Example:

```
+  LTRIM('  A B   ') yields 'A B   '
+  LTRIM('        ') yields ''
```

+ 11.6.6 COMPRESS FUNCTION

+ Remove Multiple Blanks

```
definition:

function COMPRESS(
  const   SOURCE   : STRING)
                   : STRING;

where:

SOURCE is a the STRING expression
   to be compressed.
```

+ The COMPRESS function replaces multiple blanks with a single blank.

+ Example:

```
+  COMPRESS('A B   CD   ') yields 'A B CD '
```

11.6.7 INDEX FUNCTION

Lookup String

```
definition:

function INDEX(
  const   SOURCE  : STRING;
  const   LOOKUP  : STRING)
                  : 0..255;


where:

SOURCE is a STRING that contains
   the data to be compared against.
LOOKUP is the data to be looked
  up in the SOURCE.
```

The INDEX function compares the second parameter against the first and returns the starting index of the first instance where LOOKUP begins in SOURCE. If there are no occurrences, then a zero is returned.

Examples:

```
 var
   S  : STRING;
   ...
   S := 'ABCABC';
   ...
   INDEX(S,'BC') yields 2
   INDEX(S,'X') yields 0
```

11.6.8 TOKEN PROCEDURE

Find Token

```
definition:

procedure TOKEN(
  var     POS     : INTEGER;
  const   SOURCE  : STRING;
  var     RESULT  : ALPHA);


where:

POS is the starting index in SOURCE
  of where to look for a token, it
  is set to the index of where to
  resume the search on the next
  use of TOKEN.
SOURCE is a STRING that contains
  the data from which a token
  is to be extracted.
RESULT is the variable which will
  be returned with token found.
```

The TOKEN procedure scans the SOURCE string looking for a token and returns it as an ALPHA. The starting position of the scan is passed as the first parameter. This parameter is changed to reflect the position which the scan is to be resumed on subsequent calls. Leading blanks, multiple blanks and trailing blanks are ignored. If there is no token in the string, POS is set to LENGTH(SOURCE)+1 and RESULT is set to all blanks.

A token is defined to be any of:

- Pascal/VS identifier - 1 to 16 alphanumeric characters, '$' or an underscore. The first letter must be alphabetic or a '$'.
- Pascal/VS unsigned number - see page 14.
- The following special symbols:

```
   +     -     *     /     ->    {     ¢
   =     <>    <     <=    >=    >     !
   (     )     [     ]     '     "     %
   |     &     &&    ||    ¬     ¬=    #
   :     ;     :=    .     ,     ..
   {     }     (*    *)    /*    */
```

+ Example:

```
+  I := 2;
+  TOKEN(I,', Token+', RESULT)

+    I is set to 8
+    RESULT is set to 'Token        '

+    TOKEN would return the same if
+    I were set to 3, that is,
+    leading blanks are ignored.
```

11.7 GENERAL ROUTINES

These routines provide several useful features of the Pascal/VS runtime environment.

+ 11.7.1 TRACE PROCEDURE

Routine Trace

```
definition:

procedure TRACE;
```

+ This procedure displays the current list of procedures and functions that are pending execution (i.e. save chain). The statement numbers of the statement that contained the call are also displayed. The information is written to OUTPUT.

+ 11.7.2 HALT PROCEDURE

Halt Program Execution

```
definition:

procedure HALT;
```

+ This routine halts execution of an Pascal/VS program. That is, this can be considered to be a return from the main program.

11.8 SYSTEM INTERFACE ROUTINES

These routines provide interfaces to system facilities: in general they are dependent on the implementation of Pascal/VS.

+ 11.8.1 DATETIME PROCEDURE

+ Get Date and Time

```
definition:

procedure DATETIME(
     var DATE,
         TIME : ALFA);

where:

DATE is the returned date.
TIME is the returned time.
```

+ This procedure returns the current date and time of day as two ALFA arrays. The format of the result is placed in the first and second parameters respectively:

```
mm/dd/yy
HH:MM:SS
```

+ where:
- mm is the month expressed as a two digit value.
- dd is the day of the month.
- yy is the last two digits of the year.
- HH is the hour of the day expressed in a 24 hour clock.
- MM is the minute of the hour.
- SS is the second of the minute.

+ 11.8.2 CLOCK FUNCTION

+ Get Execution Time

```
definition:

function CLOCK : INTEGER;
```

+ The value returned is the number of microseconds the program has been running. Note: In an MVS system: the time is "TASK" time; and in a CMS system: the time is "CPU virtual" time.

+ 11.8.3 PARMS FUNCTION

Get Execution Parameters

```
definition:

function PARMS : STRING;
```

The PARMS function returns a string that was associated with initial invocation of the Pascal/VS main program.

+ 11.8.4 RETCODE PROCEDURE

Set Program Return Code

```
definition:

procedure RETCODE(
        RETVALUE : INTEGER);

where:

RETVALUE is the return code to be
  passed to the caller of the
  Pascal/VS program. The value
  is system dependent.
```

The value of the operand will be returned to system when an exit is made from the main program. If this routine is called several times, only the last value specified will be passed back to the system.

12.0 THE % FEATURE

```
Syntax:

include-statement:

---> % ---> INCLUDE ---> id ---┬--------------------------------┬---------->
                               └---> ( ---> id ---> ) --->┘

check-statement:

---> % ---> CHECK ---┬----------------------┬----┬---> ON -----┬---------->
                     ├---> POINTER ------>┤    └---> OFF --->┘
                     ├---> SUBSCRIPT --->┤
                     ├---> SUBRANGE ---->┤
                     ├---> FUNCTION ---->┤
                     └---> CASE -------->┘

print-statement:

---> % ---> PRINT ---┬---> ON -----┬------------------------------------>
                     └---> OFF --->┘

list-statement:

---> % ---> LIST ---┬---> ON -----┬------------------------------------>
                    └---> OFF --->┘

page-statement:

---> % ---> PAGE ------------------------------------------------------>

title-statement:

---> % ---> TITLE ---> any-character-string ------------------------->

skip-statement:

---> % ---> SKIP ---> unsigned-integer ------------------------------>
```

The % feature of Pascal/VS is used to enable or disable a number of compiler options and features. The compiler treats a % command as a trigger symbol which causes the compiler to ignore all text between the %statement and the end-of-line.

+ 12.1 THE %INCLUDE STATEMENT

+ The INCLUDE statement provides you with
+ a means to include source code from
+ another source file or library.

+ The 'id' is the name of a file to be
+ inserted into the input stream. If an
+ identifier enclosed in parentheses
+ follows the file name, then that
+ represents a member of a library file.

+ 12.2 THE %CHECK STATEMENT

+ The CHECK statement gives you the abil-
+ ity to enable or disable the runtime
+ checking features of Pascal/VS. The
+ checking may be enabled for part or all
+ of the program. The compiler will check
+ the following:

+ • use of a pointer whose value is NIL
+ (POINTER).

+ • use of a subscript which is out of
+ range for the array index (SUB-
+ SCRIPT).

+ • lack of an assignment of a value to
+ a function before exiting from the
+ function (FUNCTION).

+ • assignment of a value which is not
+ in the proper range for the target
+ variable (SUBRANGE).

+ • use of the predefined functions
+ PRED or SUCC where the result of
+ the function is not a value in the
+ type, i.e. underflow or overflow of
+ the value range (SUBRANGE).

+ • the value of a CASE statement
+ selector which is not equal to any
+ of the CASE labels (CASE).

+ If the check option is missing, then
+ all of the above checks will be assumed
+ applicable. For example, '%CHECK ON'
+ activates all of the checks. '%CHECK
+ POINTER OFF' will disable the check on
+ pointer references. The default is:

+ % CHECK ON

+ The %CHECK statement, like the other
+ statements in this section, is a
+ direction to the compiler. Its effect
+ is based on where it appears in the
+ text and is not subject to any struc-
+ turing established by the program.

+ 12.3 THE %PRINT STATEMENT

+ The PRINT statement is used to turn on
+ and off the printing of source in the
+ listing. The default is:

+ % PRINT ON

+ 12.4 THE %LIST STATEMENT

+ The LIST statement is used to enable or
+ disable the pseudo-assembler listing
+ of the Pascal/VS translator. The
+ default is:

+ % LIST OFF

+ 12.5 THE %PAGE STATEMENT

+ The PAGE statement is used to force a
+ skip to the next page on the output
+ listing of the source program.

+ 12.6 THE %TITLE STATEMENT

+ The TITLE statement is used to set the
+ title in the listing. It also causes a
+ page skip. The title is printed as
+ specified on the statement, there is no
+ change from lower case to upper case.
+ The default is no title.

+ 12.7 THE %SKIP STATEMENT

+ The SKIP statement is used to force one
+ or more blank lines to be inserted into
+ the source listing.

A.0 THE SPACE TYPE

A.1 THE SPACE DECLARATION

```
Syntax:

space-type:

---> space ---> [ --->{constant-expr}---> ] ---> of --->{type}------------->
```

The need arises to represent data within storage areas which do not have the same fixed offset within each instance of the area. Examples of this include entries within a directory, where each entry may be of variable length, and processing variable length records from a buffer. To solve this problem, Pascal/VS provides the space structure.

A variable declared with the space type has a component which is able to 'float' over a storage area in a byte oriented manner. Space variables are accessed by following the variable's name with an integer index expression enclosed in square brackets. The index represents the offset (in bytes) within the space storage where the data to be accessed resides. The offset is specified with an origin of zero.

The constant expression which follows the space qualifier in the type definition represents the size of the storage area (in bytes) associated with the type.

The component type of the space may be of any type except a file type.

An element of a space may not be passed as a var parameter to a routine. However, an element may be passed as a const or value parameter.

A.2 SPACE REFERENCING

A component of a space is selected by placing an index expression, enclosed within square brackets, after the space variable (just as in array references). The indexing expression must be of type INTEGER (or a subrange thereof). The value of the index is the offset within the space at which the component is to be accessed. The unit of the index is the byte. The index is always based upon a zero origin. The component will be of the space base type.

If the '%CHECK SUBSCRIPT' option is enabled, the index expression will be checked at execution time to make sure that the computed address does not lie outside the storage occupied by the space. An execution time error diagnostic will occur if the value is invalid. (For a description of the CHECK feature see section 12.2 on page 134.)

```
var
  S: space[100] of
       record
         A,B: INTEGER
       end;

begin
  (*base record begins
    at offset 10 within
    space *)
  S[10].A := 26;
  S[10].B := 0;
end;
```

Space Referencing Examples

B.0 STANDARD IDENTIFIERS IN PASCAL/VS

A standard identifier is the name of a constant, type, variable or routine that is predefined in Pascal/VS. The name is declared in every module prior to the start of your program. You may redefine the name if you wish; however, it is better to use the name according to its predefined meaning.

The identifiers that are predefined are:

Standard Identifiers

	identifier	form	description
	ABS	function	compute the absolute value of an INTEGER or REAL
+	ALFA	type	array of 8 characters, indexed 1..ALFALEN
+	ALFALEN	constant	HBOUND of type ALFA, value is 8
+	ALPHA	type	array of 16 characters, indexed 1..ALPHALEN
+	ALPHALEN	constant	HBOUND of type ALPHA, value is 16
	ARCTAN	function	returns the arctangent of the argument
	BOOLEAN	type	data type composed of the values FALSE and TRUE
	CHAR	type	character data type
	CHR	function	convert an integer to a character value
+	CLOCK	function	returns the number of micro seconds of execution
+	CLOSE	procedure	close a file
+	COLS	function	returns current column on output line
+	COMPRESS	function	replaces multiple blanks in a string with one blank
	COS	function	returns the cosine of the argument
+	DATETIME	procedure	returns the current date and time of day
+	DELETE	function	returns a string with a portion removed
	DISPOSE	procedure	deallocate a dynamic variable
	EOF	function	test file for end of file condition
	EOLN	function	test file for end of line condition
	EXP	function	returns the base of the natural log (e) raised to the power of the argument
	FALSE	constant	constant of type BOOLEAN, FALSE < TRUE
+	FLOAT	function	convert an integer to a floating point value
	GET	procedure	advance file pointer to next element of input file
+	HALT	procedure	halts the programs execution
+	HBOUND	function	determine the upper bound of an array
+	HIGHEST	function	determine the maximum value of a scalar
+	INDEX	function	looks up one string in another
	INPUT	variable	default input file
	INTEGER	type	integer data type
+	INTERACTIVE	procedure	open a file for interactive input
+	LBOUND	function	determine the lower bound of an array
+	LENGTH	function	determine the current length of a string
	LN	function	returns the natural logarithm of the argument
+	LOWEST	function	determine the minimum value of a scalar
+	LTRIM	function	returns a string with leading blanks removed
+	MARK	procedure	routine to create a new heap
+	MAX	function	determine the maximum value of a list of scalars
	MAXINT	constant	maximum value of type INTEGER

	Standard Identifiers Continued		
	identifier	form	description
+	MIN	function	determine the minimum value of a list of scalars
+	MININT	constant	minimum value of type INTEGER
	NEW	procedure	allocate a dynamic variable from most recent heap
	ODD	function	returns TRUE if integer argument is odd
+	OPEN	procedure	routine to open and initialize a file
	ORD	function	convert a scalar value to an integer
	OUTPUT	variable	default output file
	PACK	procedure	copies an array to a packed array
	PAGE	procedure	skips to the top of the next page
+	PARMS	function	returns the system dependent invocation parameters
+	POINTER	type	type to permit passing arbitrary pointers a routine
	PRED	function	obtain the predecessor of a scalar
	PUT	procedure	advance file pointer to next element of output file
+	RANDOM	function	returns a pseudo-random number
	READ	procedure	routine to read data from a file
	READLN	procedure	routine to read the end of line character of TEXT file
	REAL	type	floating point
+	RELEASE	procedure	routine to destroy one or more heaps
	RESET	procedure	open a file for input
+	RETCODE	procedure	sets the system dependent return code
	REWRITE	procedure	open a file for output
	ROUND	function	convert a floating point to an integer by rounding
	SIN	function	returns the sine of the argument
+	SIZEOF	function	determine the memory size of a variable or type
	SQRT	function	returns the square root of the argument
	SQR	function	returns the square of the argument
+	STR	function	convert an array of characters to a string
+	SUBSTR	function	returns a portion of a string
	SUCC	function	obtain the successor of a scalar
	TEXT	type	file of CHAR
+	TOKEN	procedure	extracts tokens from a string
+	TRACE	procedure	writes the routine return stack
+	TRIM	function	returns a string with trailing blanks removed
	TRUE	constant	constant of type BOOLEAN, TRUE > FALSE
	TRUNC	function	convert a floating point to an integer by truncating
	UNPACK	procedure	copies a packed array to an array
	WRITE	procedure	routine to write data to a file
	WRITELN	procedure	routine to write end of line to a TEXT file

C.0 SYNTAX DIAGRAMS

actual-parameters:

```
        ───────┬───> ( ───┬──────────>{expr}─────┬──────> ) ───┬──────────────────────────────>
               │          │    └<────── , <──────┘ │            │
   +           │          └--------------------->┘            │
               └────────────────────────────────────────────>┘
```

array-structure:

```
   +    --->{id:type}---> ( --->┐
   +       ┌<--------------------┘
   +       └--┬----┬--->{constant-expr}--┬--> : {repetition}--┬--┬----┬---> ) ---------->
   +          │    │                     └-------------------->┘  │    │
   +          │    └-------------------------------------------->┘    │
   +          └<---------------- , <--------------------------------┘
```

array-type:

```
        ───────┬───> packed ───┐
               ├<──────────────┘
               └──────────> array [ ──┬────>{index-type}───┬───> ] of ───>{type} ──────>
                                       └<────── , <─────────┘
```

assert-statement:

```
   +    ---> assert --->{expr}----------------------------------------------------------->
```

assignment-statement:

```
        ──────┬───>{variable}────────────┬────> := ───>{expr}──────────────────────────────>
              └───>{id:function}────>┘
```

base-scalar-type:

```
               ┌───>{enumerated-scalar-type}───>┐
        ───────┼───>{id:scalar-type}───────────>┼──────────────────────────────────────────>
               └───>{subrange-scalar-type}─────>┘
```

case-statement:

```
──> case ──>{expr}──> of ──>┐
     ┌<─────────────────────┘
     │
     ├─────────┬───>{range}──┬───> : ──>{statement}───┬──>┐
     │    │    └<──── , <────┘                        │   │
     │    └<────────────────────── ; <────────────────┘   │
     ├<───────────────────────────────────────────────────┘
     │                         └<── ; <──┘
+    ├---> otherwise ---┬--->{statement}---┬--->┐
+    │                  └<------- ; <------┘    │
+    ├<-----------------------------------------┘
     │
     └──> end ───────────────────────────────────────────────>
```

check-statement:

```
---> % ---> CHECK --┬--------------------┬---┬---> ON ----┬------------------>
                    ├---> POINTER ------>┤   └---> OFF --->┘
                    ├---> SUBSCRIPT --->─┤
                    ├---> SUBRANGE ---->─┤
                    ├---> FUNCTION ---->─┤
                    └---> CASE -------->─┘
```

compound-statement:

```
──> begin ──┬───>{statement}──┬──> end ──────────────────────────>
            └<────── ; <──────┘
```

constant:

```
───┬───>{unsigned-constant}──────────────────────────────────────>
   └───┬──> + ───┬───>{unsigned-number}───>┘
       └──> - ──>┘
```

constant-dcl:

```
+    ──> const ──┬───>{id}──> = ──>{constant-expr}──> ; ──┬──────────────>
                 └<───────────────────────────────────────┘
```

continue-statement:

```
+    ---> continue ------------------------------------------------------>
```

declaration:

```
  ──────┬────>{label-dcl}────────>┐
        ├────>{constant-dcl}─────>┤
        ├────>{type-dcl}─────────>┤
        ├────>{var-dcl}──────────>┤
+       ├---->{def-dcl}---------->┤
+       ├---->{static-dcl}------->┤
+       ├---->{value-dcl}-------->┤
+       │                         │
+       └────>{routine-dcl}──────>┴──────────────────────────────────────>
```

def-dcl:

```
+ ---┬----> def ---┬----------┬----┬--->{id}---┬---> : --->{type}---> ; ---┬----------->
+    └----> ref --->┘          │    └<---  ,  ----┘                          │
+                              │                                            │
+                              └<-------------------------------------------┘
```

empty-statement:

```
──────────────────────────────────────────────────────────────────────────>
```

enumerated-scalar-type:

```
───> ( ───┬───>{id}───┬────> ) ─────────────────────────────────────────>
          └<───  ,  <─┘
```

expr:
constant-expr:

```
───>{simple-expression}────┬──────────────────────────────────────────────┬────>
                           ├───> =   ──────────>{simple-expression}───>───┘
                           ├───> <>  ───>┤
                           ├───> <   ───>┤
                           ├───> <=  ───>┤
                           ├───> >=  ───>┤
                           ├───> >   ───>┤
                           └───> in  ───>┘
```

factor:

```
  ──────┬────>{function-call}────────────────────────────────┬──────────>
        ├────>{variable}────────────────────────────────────>┤
        ├────>{set-factor}──────────────────────────────────>┤
        ├────> ( ───>{expr}───> ) ──────────────────────────>┤
+       ├---->{structured-constant}------------------------->┤
        ├────> not ───>{factor}─────────────────────────────>┤
        └────>{unsigned-constant}───────────────────────────>┘
```

field:

```
──>{id}──┬─────────────────────────────────────────────┬──────────────────>
         └---> ( --->{constant-expr}---> ) --->────────┘
```

field-list:

```
──┬───>{fixed-part}───┬──────> ; ──┬───>{variant-part}──┬──┬──> ; ──┬──>
  │                   └────────────┼────────────────────┘  └────────┘
  └────────────────────────────────┘
```

file-type:

```
──> file of ──>{type}──────────────────────────────────────────────────>
```

fixed-part:

```
────┬────────┬──┬──>{field}──┬─────┬───> : ───>{type}───┬──────────────>
    │        │  └<──── , <───┘     │                    │
+   │        └---------------------┘                    │
    └<──────────────── ; <──────────────────────────────┘
```

for-statement:

```
──> for ──>{id}──> := ──>{expr}──┬──> to ─────┬──>{expr}──>┐
                                 └──> downto ─┘            │
                     ┌<────────────────────────────────────┘
                     │
                     └─────────> do ──>{statement}─────────────────────>
```

formal:

```
────┬────┬──> var ─────┬──┬───{id}───┬────> : ──>{id:type}──────┬────>
+   │    └---> const --┘  └<──── , <─┘                          │
    │                                                           │
    ├────┬──{id}───┬────> : ────>{id:type}─────────────────────>┤
    │    └<─── , <─┘                                            │
    ├───>{procedure-heading}───────────────────────────────────>┤
    └───>function-heading}─────────────────────────────────────>┘
```

formal-parameters:

```
────┬────> ( ──┬──>{formal}──┬───> ) ──┬───────────────────────────────>
    │          └<──── ; <────┘         │
    └─────────────────────────────────>┘
```

function-call:

```
──>{id:function}──>{actual-parameters}─────────────────────────────────>
```

function-heading:

```
——> function ——>{id}——>{formal-parameters}——> : ——>{id:type}——————————>
```

goto-statement:

```
—————————> goto ——>{label}————————————————————————————————————————————————>
```

id:

```
                                   ┌──────────────────────────────────────────>
                                   │
                                   ├——>{digit}————————>┐
——>{letter}—————┬──────────────────┼——>{letter}———————>┼———>┐
                │                  └--->{underscore}-->┘    │
                │                                           │
                └<──────────────────────────────────────────┘
```

if-statement:

```
 —> if —>{expr}—> then —>{statement}———————————————————————————————>
                                    │                          │
                                    └——> else ——>{statement}——>┘
```

include-statement:

```
---> % ---> INCLUDE ---> id ---┬-----------------------------┬----------------->
                               └---> ( ---> id ---> ) --->┘
```

index-type:

```
       ┌——>{enumerated-scalar-type}———>┐
———————┼——>{id:scalar-type}———————————>┼————————————————————————————————————>
       └——>{subrange-scalar-type}—————>┘
```

label:

```
———————>{unsigned-integer}———┬————————————————————————————————————————————>
       └---->{id}---------------->┘
```

label-dcl:

```
——> label ————┬————>{label}————┬——> ; ——————————————————————————————————>
              └<————— , <——————┘
```

leave-statement:

```
+ ---> leave ----------------------------------------------------------------->
```

list-statement:

```
---> % ---> LIST ---┬---> ON -----┬--------------------------------------->
                    └---> OFF --->┘
```

module:

```
───┬──>{program-module}──┬──────────────────────────────────────────>
   └--->{segment-module}--->┘
```

page-statement:

```
---> % ---> PAGE ------------------------------------------------------>
```

pointer-type:

```
──> -> ──>{id:type}─────────────────────────────────────────────────>
```

print-statement:

```
---> % ---> PRINT ---┬---> ON ----┬------------------------------------------>
                     └---> OFF --->┘
```

procedure-call:

```
──>{id:procedure}──┬───────────────────────────────────┬──────────────>
                   └──> ( ──┬──>{expr}──┬──> ) ──>┘
                            └<── , <────┘
```

procedure-heading:

```
──> procedure ──>{id}──>{formal-parameters}─────────────────────────>
```

program-module:

```
──> program ──>{id}──┬───> ( ──┬──>{id}──┬──> ) ──┬──>┐
                     │         └<── , <──┘         │   │
                     └-------------------------------->┘   │
    ┌<───────────────────────── ; <──────────────────────┘
    ├<─────────────────┐
    ├──>{declaration}──>┘
    └──>{compound-statement}──> . ──────────────────────────────>
```

range:

```
+   ──>{constant-expr}──┬─────────────────────────────────────────────>
+                       └---> .. --->{constant-expr}--->┘
```

real-number:

```
+   ───┬---> ' ---┬--->{hex-digit}---┬---> 'XR ----------------->┐
+      │          └<-----------------┘                            │
       └──┬──>{digit}──┬─────┬──> . ──┬──>{digit}──┬──┬──────┴──────>
          └<───────────┘     │        └<───────────┘  │      │
          ├<─────────────────┘                        │      │
          ├<──────────────────────────────────────────┘      │
          └──> E ──┬────────>┬──┬──>{digit}──┬──────────>┘
                   ├──> + ──>┤  └<───────────┘
                   └──> - ──>┘
```

record-structure:

```
+    --->{id:type}---> ( ---┬----┬--->{constant-expr}---┬----┬---> ) ------------>
+                           │    └---------------------->┘    │
+                           └<------------ , <--------------┘
```

record-type:

```
────┬──────────────────┬──> record ──>{field-list}──> end ──────────────>
    └──> packed ──>┘
```

repeat-statement:

```
──> repeat ────────┬──>{statement}───┬───> until ──>{expr}──────────────>
                   └<──────── ; <────┘
```

repetition:

```
+    --->{constant-expr}------------------------------------------------->
```

return-statement:

```
+    ---> return --------------------------------------------------------->
```

routine-dcl:

```
─────┬──>{procedure-heading}─────────────────>┐
     └──>{function-heading}──────────────────>┤
                                              │
     ┌<──────────────── ; <───────────────────┘
     │
+    ├---> EXTERNAL ---> ; ------------------------------->┐
+    ├---> FORTRAN ----> ; ------------------------------->┤
+    ├---> ENTRY ---> ; --->┐                              │
+    ├<---------------------┘                              │
     ├──> FORWARD ──> ; ──────────────────────────────────>┤
     │                                                     │
     ├<─────────────────────┐                              │
     ├──>{declaration}───>┘                                │
     │                                                     │
     └──>{compound-statement}──> ; ───────────────────────┴──────────>
```

segment-module:

```
+    ---> segment --->{id}---> ; --->┐
+                                     │
+      ┌<---------------------------┬-┘
+      │                            │
+      ├--->{constant-dcl}---->─────┤
+      │                            │
+      ├--->{type-dcl}--------->────┤
+      │                            │
+      ├--->{def-dcl}---------->────┤
+      │                            │
+      ├--->{static-dcl}------>─────┤
+      │                            │
+      ├--->{value-dcl}------->─────┤
+      │                            │
+      ├--->{routine-dcl}----->─────┘
+      │
+      └---> . -------------------------------------------------------------->
```

set-factor:

```
──> [ ──────────────>{expr}──┬──────────────────────────┬──────────┬──> ] ──────>
          │   ┌              └──> .. ──>{expr}──>┘       │          │
          │   └<──────────────────── , <────────────────┘          │
          └─────────────────────────────────────────────────────>──┘
```

set-type:

```
──┬────────────────────────┬──> set of ──>{base-scalar-type}──────────────────────>
  └──> packed ──────>──────┘
```

simple-expression:

```
──┬───────────────────┬─┬─────>{term}──┬─────────────────────────────────────────>
  ├──> + ───>─────────┤ │              ├──> + ──>┐
  └──> - ───>─────────┘ │              ├──> - ──>┤
+                       │              ├---> && -->┤
                        │              └──> | ──>┤
                        └<───────────────────────┘
```

skip-statement:

```
---> % ---> SKIP ---> unsigned-integer ------------------------------------->
```

space-type:

```
+    ---> space ---> [ --->{constant-expr}---> ] ---> of --->{type}------------->
```

statement:

```
        ──────┬────>{label}───> : ───>┐
              ├<──────────────────────┘
              │
  +           ├--->{assert-statement}------------------------------------>┐
              ├───>{assignment-statement}────────────────────────────────>┤
              ├───>{case-statement}──────────────────────────────────────>┤
              ├───>{compound-statement}──────────────────────────────────>┤
  +           ├--->{continue-statement}---------------------------------->┤
              ├───>{empty-statement}─────────────────────────────────────>┤
              ├───>{for-statement}───────────────────────────────────────>┤
              ├───>{goto-statement}──────────────────────────────────────>┤
              ├───>{if-statement}────────────────────────────────────────>┤
  +           ├--->{leave-statement}------------------------------------->┤
              ├───>{procedure-call}──────────────────────────────────────>┤
              ├───>{repeat-statement}────────────────────────────────────>┤
  +           ├--->{return-statement}------------------------------------>┤
              ├───>{while-statement}─────────────────────────────────────>┤
              └───>{with-statement}──────────────────────────────────────>┴──────────>
```

static-dcl:

```
  +     ---> static ---┬---┬--->{id}---┬---> : --->{type}---> ; ---┬---------------->
  +                    │   └<--- , ----┘                           │
                       │                                           │
  +                    └<------------------------------------------┘
```

string:

```
        ──────┬──> ' ─────┬──────────────────────┬───> ' ──────┬──────────────────>
              │           └<──{character}<───────┘              │
  +           └---> ' ---┬--->{hex-digit}---┬---> 'XC ------>┘
  +                      └<-----------------┘
```

string-type:

```
  +     ---> STRING ---┬---> ( --->{constant-expr}---> ) ---┬---------------------->
  +                    └------------------------------------>┘
```

structured-constant:

```
  +     ---┬--->{record-structure}---┬---------------------------------------->
  +        └--->{array-structure}--->┘
```

subrange-scalar-type:

```
+    ──┬---> packed --->┐
       ├<---------------┘
+      └──┬─>{constant}──> .. ──>{constant-expr}──────────────────────┬──>
+         └---> range --->{constant-expr}---> .. --->{constant-expr}--->┘
```

term:

```
     ──┬──>{factor}──┬────────────────────────────────────────────────>
       │             ├──> *  ───>┐
       │             ├──> /  ───>┤
       │             ├──> div ──>┤
       │             ├──> mod ──>┤
+      │             ├---> >> --->┤
+      │             ├---> << --->┤
+      │             ├---> || --->┤
       │             └──> &  ───>┤
       └<────────────────────────┘
```

title-statement:

```
---> % ---> TITLE ---> any-character-string ------------------------------->
```

type:

```
──┬──>{id:type}──────────────────────────────────────>┐
  ├──>{enumerated-scalar-type}───────────────────────>┤
  ├──>{subrange-scalar-type}─────────────────────────>┤
  ├──>{array-type}───────────────────────────────────>┤
  ├──>{record-type}──────────────────────────────────>┤
  ├──>{set-type}─────────────────────────────────────>┤
  ├──>{file-type}────────────────────────────────────>┤
  └──>{pointer-type}─────────────────────────────────>┴────>
```

type-dcl:

```
──> type ──┬──>{id}──> = ──>{type}──> ; ──┬──────────────────>
           └<─────────────────────────────┘
```

unsigned-constant:

```
──┬──>{unsigned-number}──┬────────────────────────────────>
  ├──>{string}─────────>┤
  ├──>{id:constant}────>┤
  └──> nil ────────────>┘
```

unsigned-integer:

```
      ──────┬──┬──>{digit}──┬──────────────────────────────────────────>
            │  └<───────────┘                                      │
  +         ├---> ' ---┬---->{binary digit}---┬---> 'B --->┤
  +         │          └<---------------------┘            │
  +         └---> ' ---┬---->{hex-digit}---┬---> 'X ------>┘
  +                    └<------------------┘
```

unsigned-number:

```
      ──────┬────>{unsigned-integer}───>┬────────────────────────────────>
            └────>{real-number}────────>┘
```

value-assignment:

```
  +   --->{variable}---> := ---┬---->{constant-expression}----┬-------------------->
  +                            └---->{structured-constant}--->┘
```

value-dcl:

```
  +   ---> value ---┬---{value-assignment}---> ; ---┬----------------------------->
  +                 └<------------------------------┘
```

var-dcl:

```
      ───> var ────┬───┬──>{id}────┬────> : ───>{type}───> ; ───┬────────────────>
                   │   └<──── , ───┘                            │
                   └<───────────────────────────────────────────┘
```

variable:

```
      ───>{id}───>┐
                  ├<────────────────────────────────────────┐
                  │                                         │
                  ├────> [ ────┬──>{expr}────┬────> ] ─────>┤
                  │            └<──── , ─────┘              │
                  ├────> . ───>{id:field}──────────────────>┤
                  ├────> -> ───────────────────────────────>┘
                  │
                  └─────────────────────────────────────────────────────────>
```

variant-part:

```
  +                                      ┌------------------->┐
      ───> case ────┬────>{field}───> : ─┴───┬──>{id:type}────┴───> of ───>┐
                    └────────────────────>───┘                              │
          ┌<────────────────────────────────────────────────────────────────┘
          │
          └───┬──┬──>{range}────┬────> : ( ────┬───>{field-list}────┬───> ) ───┬──────>
              │  └<──── , <─────┘              └───────────────────>┘          │
              └<──────────────────────────── ; <───────────────────────────────┘
```

while-statement:

```
      ────> while ───>{expr}───> do ───>{statement}──────────────────────────────>
```

with-statement:

```
——> with ——┬——>{variable}——┬——> do ——>{statement}——————————————————>
           └<———————— , <——————┘
```

D.0 INDEX TO SYNTAX DIAGRAMS

E.0 GLOSSARY

Actual parameter specifies what is to be passed to a routine.

Array type is the structured type that consists of a fixed number of elements, each element of the same type.

Assignment compatible is the term used to indicate whether a value may be assigned to a variable.

Automatic variable is a variable which is allocated on entry to a routine and is deallocated on the subsequent return. An automatic variable is declared with the var declaration.

Base scalar type is the name of the type on which another type is based.

Bit is one binary digit.

Byte is the unit of addresability on the System/370, its length is eight bits.

Compatible types is the term which is used to indicate that operations between values of those types are permited.

Component is the name of a value in a structured type.

Constant is a value which is either a literal or an identifier which has been associated with a value in a const declaration.

Constant expression is an expression which can be completely evaluated by the compiler at compile time.

Dynamic variable is a variable which is allocated under programmer control. Explicit allocates and deallocates are required; the predefined procedures NEW and DISPOSE are provided for this purpose.

Element is the component of an array.

Entry routine is a procedure or function which may be invoked from outside the module in which it is defined. The routine is called entry in the module in which is defined. An entry routine may not be imbedded in another routine; it must be defined on the outermost level of a module.

Enumerated scalar type is a scalar that is defined by enumerating the elements of the type. Each element is represented by an identifier.

External routine is a procedure or function which may be invoked from outside the module in which the routine is defined.

Field is the component of a record.

File type is a data type which is the mechanism to do input and output in Pascal/VS.

Fixed part is that part of a record which exists in all instances of a particular record type.

Formal parameter is a parameter as declared on the routine heading. A formal parameter is used to specify what is permitted to be passed to a routine.

Function is a routine, invoked by coding its name in an expression, which passes a result back to the invoker through the routine name.

Identifier is the name of a declared item.

Index is the selection mechanism applied to an array to identify an element of the array.

Internal routine is a routine which can be used only from within the lexical scope in which it was declared.

Lexical scope identifies the portion of a module in which a name is known. An identifier declared in a routine is known within that routine and within all nested routines. If a nested routine declares an item with the same name, the outer item is not available in the nested routine.

Module is the compilable unit in Pascal/VS.

Offset is the selection mechanism of a space. An element is selected by placing an integer value in parenthesis. The origin of a space is based on zero.

Packed record type is a record structure in which fields are allocated in the minimum number of bytes. Implementation defined alignment of data types will not be preserved in order to pack the record. Packed records may not be passed by read/write reference.

Pass by read only reference is the parameter passing mechanism by which the address of a variable or temporary is passed to the called routine. The called routine is not permitted to modify the formal parameter. If the actual parameter is an expression, a temporary will be created and its address will be passed to the called routine. A temporary is also created for fields of packed records.

Pass by read/write reference is the parameter passing mechanism by which the address of a variable is passed to the called routine. If the called routine modifies the formal parameter, the corresponding actual parameter is changed. Only variables may be passed via this means. Fields of packed records will not be permitted to be passed in this way.

Pass by value is the parameter passing mechanism by which a copy of the value of the actual parameter is passed to the called routine. If the called routine modifies the formal parameter, the corresponding actual parameter is not affected.

Pointer type is used to define variables that contain the address of dynamic variables.

Procedure is a routine, invoked by coding its name as a statement, which does not pass a result back to the invoker.

Program module is the name of the compilable unit which represents the first unit executed.

Record type is the structured type that contains a series of fields. Each field may be of a type different from the other fields of the record. A field is selected by the name of the field.

Reserved word is an identifier whose use is restricted by the Pascal/VS compiler.

Routine is a unit of a Pascal/VS program that may be called. The two type of routines are: procedures and functions.

Scalar type defines a variable that may contain a single value at execution.

Segment module is a compilable unit in Pascal/VS that is used to contain entry routines.

Set type is used to define a variable that represents all combinations of elements of some scalar type.

Space type is used to define a variable whose components may be positioned at any byte in the total space of the variable.

Statement is the executable unit in a Pascal/VS program.

String represents an ordered list of characters whose size may vary at execution time. There is a maximum size for every string.

String constant is a string whose value is fixed by the compiler.

Structured type is any one of several data type mechanisms that defines variables that have multiple values. Each value is referred to generally as a component.

Subrange scalar type is used to define a variable whose value is restricted to some subset of values of a base scalar type.

Tag field is the field of a record which defines the structure of the variant part.

Type defines the permissible values a variable may assume.

Type definition is a specification of a data type. The specification may appear in a type declaration or in the declaration of a variable.

Type identifier is the name given to a declared type.

Variant part is that portion of a record which may vary from one instance of the record to another. The variant portion consists of a series of variants that may share the same physical storage.

This page intentionally left blank.