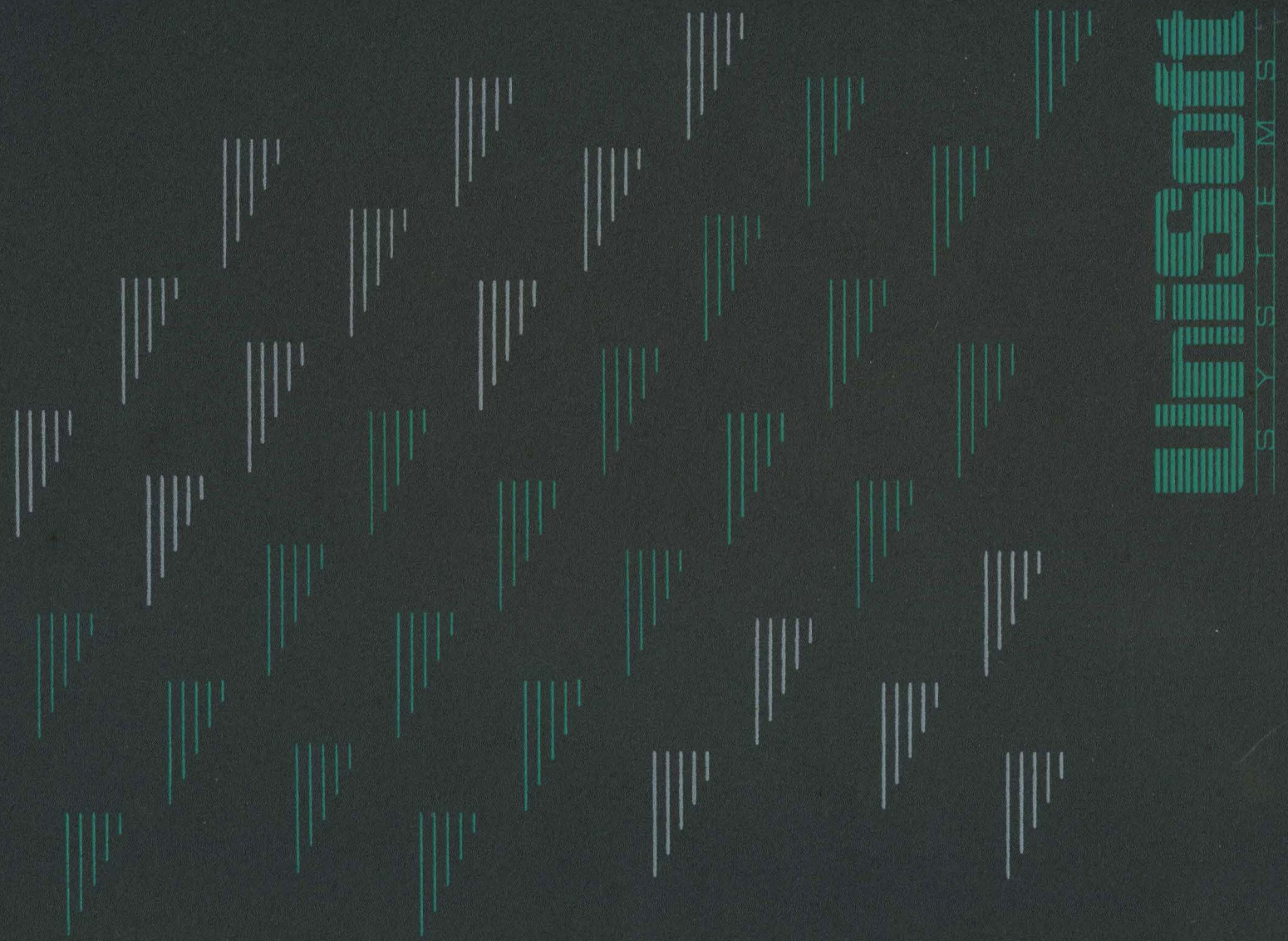


UniSoft Systems

Pascal

Language Manual for UniPlus+

UniSoft

Pascal

Language Manual for UniPlus+

UniSoft SYSTEMS

739 Allston Way, Berkeley, CA 94710
(415) 644-1230 • TWX II 910 366-2145
UUCP ucbvax!unisoft!unisoft

Pascal

Language Reference Manual

Version 2.0, 1st September 1983

PN: 1020-04

UniSoft
SYSTEMS

This **Pascal** Reference Manual was produced by:

J. Barth, R.S. Glanville, H. McGilton and M.A. Finnerty.

UniSoft Part Number: 1020-04

CONTENTS

Preface

This Pascal Reference Manual describes the Pascal Programming language as implemented by Silicon Valley Software, Inc. Throughout this manual, "SVS Pascal" is to mean that version of Pascal as implemented by Silicon Valley Software, Inc.

SVS Pascal implements the Pascal language as defined in the proposed ISO Standard that appeared in Pascal News, Number 20, December 1980. Appendix C — "Relationships to ISO Pascal" describes areas where SVS Pascal deviates from the ISO standard.

In common with many Pascal implementations, SVS Pascal has extensions. These mainly derive from features implemented in the UCSD Pascal System. Primarily, those extensions revolve around facilities for compiling code modules separately and string handling. The other major areas of extension are concerned with input and output facilities, single and double precision floating point, and with standard procedures and functions. Differences from UCSD Pascal are noted in Appendix D — "Relationships to UCSD Pascal".

Scope of this Manual

This manual is a reference manual for SVS Pascal. It is not intended as a user manual or a tutorial. Readers are expected to already have some grasp of programming concepts, terminology, and have at least a minimal understanding of Pascal. There are approximately 50 books on Pascal programming in the commercial market.

Overview of this Manual

The overall layout of this manual loosely follows that of the **Pascal User Manual and Report**, by Kathleen Jensen and Niklaus Wirth. The phrase "Jensen and Wirth" is used to refer to that book. There is somewhat more detail in this reference manual than in Jensen and Wirth.

In general, the order that topics are presented in is: first some narrative introductory material, then formal descriptions, followed by examples.

Chapter 1 – "Introduction" is an introduction to Pascal terms and concepts. It contains an overview of the Pascal language. There is a description of the metalanguage that this manual uses to describe the Pascal Language. Finally there are descriptions of the basic elements of Pascal.

Chapter 2 – "Defining Data Types" introduces the concepts of *data types* and discusses the notations by which data types are defined and declared.

Chapter 3 – "Variables" describes the means whereby *variables* are declared and referenced.

Chapter 4 – "Expressions" describes Pascal *expressions* which are used to derive new data values.

Chapter 5 – "Statements" presents Pascal *statements* and how they are used to achieve computing actions.

Chapter 6 – "Input and Output" covers Pascal *input and output* facilities.

Chapter 7 – "Program Structure" describes *Program Structure* in Pascal, including the ideas of independent compilation units.

Chapter 8 – "Standard Procedures and Functions" describes Pascal standard procedures and functions, that is, those "built in" facilities of the language that a user program need not provide.

Chapter 9 – "Pascal Compile Time Options" describes the compile time options available to the programmer, in order to exercise control over some of the actions of the Pascal compiler and the run time system.

Appendix A – "Messages from the Pascal System" is a list of diagnostic messages from the Pascal compiler and the run-time library.

Appendix B – "Pascal Language Summary" provides a summary of the Pascal language syntax.

Appendix C – "Relationships to ISO Pascal" covers the differences between SVS Pascal and ISO standard Pascal.

Appendix D – "Relationships to UCSD Pascal" covers the differences between SVS Pascal and UCSD Pascal.

Appendix E – "Data Representations" covers machine-dependent issues such as data representation, data packing and parameter passing.

Appendix F – "Operating the SVS Pascal System" describes the system independent aspects of operating the system and the considerations involved in linking programs written in several languages.

Appendix G – "Operating System Specific Information" contains a description of how to run the Pascal compiler on the host operating system, and also covers details of specific dependencies and interfacing requirements (if any) of the host operating system.

Chapter 1 – Introduction

Pascal is a "modern" computer programming language designed by Professor Niklaus Wirth (of the Eidgenossiche Technische Hocheschule, Zurich, Switzerland) in reaction to the perceived disorder of contemporary programming languages. Originally intended as an aid to teaching rigorous and disciplined computer programming, Pascal has since gained international acceptance as a programming language for a multiplicity of applications ranging from writing compilers (including Pascal compilers) to controlling a grain elevator. Pascal is not an acronym for anything. Pascal is named after Blaise Pascal, the 17th century philosopher and mathematician.

Pascal is one of the many derivatives of Algol-60. Algol introduced the notion of *nested control structures* such as **if..then..else** that form the basis of today's structured programming methods. In addition to the control structures, Pascal goes one step further with the notion that *data structures* play at least as important a part in rigorous programming as do control structures. The absence of an adequate data structuring notation was seen as Algol's most obvious deficiency.

Pascal's major contribution to the advance in programming technology is the concept of user definable data types. This provides powerful facilities for defining new data types and data structures in terms of a few basic types.

This reference manual describes the Pascal language as implemented by Silicon Valley Software, Inc. Throughout, the term "SVS Pascal" means the Pascal implementation as described in this reference manual.

1.1 Overview of the Pascal Language

A Pascal program consists of a series of *declarations* and *statements*. Declarations serve to define program objects. Statements determine actions to be performed upon such objects. These two things, declarations and statements, serve to describe a computer program.

Definable Pascal objects include *variables*, *functions*, *procedures*, and *files*. Declaring an instance of an object requires an *identifier* and, usually, a *type* description. An object's identifier serves to identify that object so that it can be referenced later. The type associated with an object defines its operational

characteristics, and in some cases, indicates a referential notation.

It is important to note that all user supplied objects must be fully described, especially as to their type. Pascal is unlike many other programming languages in that it does not supply any default attributes for undeclared identifiers.

One of Pascal's strongest points is the ability for users to define new types. Pascal supplies a small number of predefined or basic types, such as **integer**. Pascal then supplies notations for defining new (user defined) types, both in terms of the basic types, and in terms of other user defined types.

A type can be described directly in a declaration, or, a type can be referenced by a type identifier which, in turn, must be defined by another explicit type declaration.

In general, a Pascal object is only subject to operations that lie inside of a domain indicated by its type. For example, most binary operators are restricted to objects of the same type (for instance, characters and integers cannot be added directly). These operational constraints are rigid, as are the rules for type identity and assignment compatibility. Departures from the rules have to be spelled out explicitly in terms of *conversion functions.*

The basic data type is the *scalar* type, often referred to as an *enumerated* type. A scalar definition indicates an ordered set of values, where each identifier in the set stands for a specific value.

In addition to the definable scalar types, there are six *standard basic types*, namely **integer**, **longint**, **char**(acter), **real**, **double**, and **Boolean** types. With the exception of the **Boolean** type, their values are denoted by numbers or quoted characters, instead of by identifiers.

A type may also be defined as a *subrange* of a scalar type by indicating the lower and upper bounds of the subrange.

Structured types are aggregates, defined by describing the types of their components, and by indicating a *structuring method.* The structuring methods differ in the way that components of a structured variable are selected, and the operations in which they can participate. Pascal provides five basic ways to construct an aggregate object, namely **array**, **record**, **set**, **string**, and **file**.

An **array** has components which are all of the same type. A component is selected by a computable *index.* The type of such an index must be a scalar, and is determined at the time the array is declared.

A **record** has components called *fields* which need not be all of the same type. A field selector for a component of a record is an identifier that is uniquely associated with the component to be selected. Unlike an array element index, a field selector is not a computable quantity. The field selectors are defined at the same time that the record is defined. A record type may consist of several *variants.* This means that different variables of the same

record type may actually contain different structures. That is, the number and types of the components may differ between different instances of the same type. The particular variant which the specific variable assumes is indicated by a field called the *tag field*, common to all variants of that record.

A **set** is a homogeneous collection of elements selected from some *base type*. The base type might be a user defined scalar type or a subrange of some scalar type such as **integer** or **char**. A Pascal **set** is the collection of values comprising the *powerset* of the base type. That means, the set of all subsets of that base type.

A **string** data type is a sequence of characters whose length can vary dynamically during program execution. A string has a maximum length (its static length) which is determined when it is defined. There are a rich set of intrinsic procedures and functions to manipulate strings.

A **file** is a *sequence* of components of the same type. The sequence is normally associated with external storage or input and output devices, so that files are the means whereby a Pascal program communicates with the world outside of the computer. Files can be sequential such that there is a natural ordering, and only one component of the file is accessible at any one time, or they can be random, such that any given component of the file is accessible on demand.

Explicitly declared variables are called *static*, in that they are known at compile time (lexically static). A declaration associates an identifier with the variable. The identifier is subsequently used to refer to that variable. In contrast to static variables, *dynamic* variables are created by executable statements. Such a dynamic creation of a variable yields a *pointer* (which substitutes for an explicit declaration), that is subsequently used to refer to the dynamically allocated variable. Any given pointer variable may only assume values pointing to variables of a specific type, and is said to be *bound* to that type. A pointer may be assigned to other pointer variables of the same type. Any pointer can assume the value **nil** – a universal pointer that is not bound to a specific type.

The *assignment statement* is the fundamental Pascal statement. It assigns a newly computed value to a variable or a component of a variable. New values are obtained by evaluating *expressions*. Expressions consist of variables, constants, sets, operators, and functions, operating on specified objects, to produce new values. Operands of expressions are either declared in the program, or are standard Pascal entities. Pascal defines a fixed set of operators that can be considered to define a mapping from given operand types into result types. Operators encompass the four groups: (1) arithmetic operators, (2) Boolean operators, (3) set operators, and (4) relational operators.

A *procedure statement* causes execution of a designated procedure. This is known as *activating* or *calling* the procedure. Assignment and procedure statements are the basic elements of *structured statements*. Structured

statements specify sequential, selective, or repetitive execution of their component statements. Sequential execution is obtained by the *compound statement*; Conditional and selective execution by the **if** statement and the **case** statement; Repetitive execution is specified by the **while** statement, the **repeat** statement, or the **for** statement.

A statement can be given a name (an identifier), and subsequently be referenced via that name. The statement is then called a **procedure**, and its declaration is a *procedure declaration.* A procedure declaration can itself contain type declarations, variable declarations, and further procedure declarations. These subsequent declarations can only be referenced within that procedure, and are thus said to be *local* to the procedure. The program text that comprises a procedure body is called the *scope* of any identifiers declared local to that procedure. Since procedures may be declared local to other procedures, scopes may be nested. Objects declared in the main program block, not local to any procedure, are said to be *global*, in that their scope is that of the entire program.

A procedure can have a number of *parameters* (determined at procedure declaration time), each parameter being denoted by an identifier called the *formal parameter.* When a procedure is activated, each of the formal parameters has an actual quantity substituted such that that quantity is accessed by reference to the formal parameter identifier. These quantities are called *actual parameters.* There are three sorts of parameters, namely *value* parameters, *variable* parameters, and *procedure* or *function* parameters. A value parameter is an actual parameter which is evaluated once. The formal parameter then represents a local variable conveniently initialized to the value of the actual parameter. In the case of a variable parameter, the actual parameter is a variable — the formal parameter actually references and can alter that variable. Possible array indexes are evaluated before activation of the procedure or function. In the case of a procedure or function parameter, the actual parameter is a procedure or function identifier.

Functions are declared in the same way as procedures. The difference is that a function returns a value. Pascal functions have intuitive similarities to the mathematical notion — a function is a computational entity that is applied to some arguments and generates a result. Pascal functions differ from the rigorous mathematical notion of functions in that they can have side effects. The type of the returned value must be specified as part of the function declaration. Functions can only return scalar types or pointer types. A function reference must appear in the context of an expression.

Pascal procedures and functions are inherently *recursive.* That means that a procedure or function can call itself anew before the current activation is complete. On each activation, a fresh set of local data is created. Recursive activation can be direct (the reference is contained within the procedure or function itself) or indirect (the reference is from another procedure or function which in turn is referenced from the current procedure or function).

1.2 Metalanguage

A "metalanguage" is a collection of notations that describe another language. In this case the language being described is Pascal. The metalanguage used in this manual to describe Pascal is a modified version of the ubiquitous Backus-Naur Form, or BNF (first used to describe Algol). A description of the metalanguage follows.

Syntactic constructs which are enclosed between "angle brackets" < and > define the basic language elements. Every language construct should eventually be defined in terms of basic lexical constructs defined in the remainder of this chapter.

A construct appearing outside the angle brackets stands for itself, that is, it is supposed to be self denoting. Such a construct is known as a *terminal symbol.* Terminal symbols and reserved words appear in **bold face text** throughout this manual.

The symbol ::= is to be read "defined as".

The symbol .. means "through", indicating an ordered sequence of things where only the start and end elements are specified. (The reader is left to infer the middle elements). For example, the notation 'a' .. 'z' means "the ordered collection starting with the letter 'a', ending with the letter 'z', and containing the letters 'b', 'c'....'x', 'y' in between". In other words, all the lower case letters.

The "vertical bar" symbol | is read as "or". It separates sequences of elements that represent a choice of one out of many.

The metalanguage construct {...} (elements inside braces) enclose elements which are to be repeated "zero to many times". Although the braces are also used as one of the forms of comment delimiters in Pascal, this should not cause any ambiguity. The one case where ambiguity would occur is in the definition of comments, and this is explicitly pointed out at that time.

It is recognized that the syntactic descriptions are not completely rigorous in that they do not cover semantic issues. For example, the syntactic definition of a decimal number does not mention how big a number can be. Where the formal descriptions fall short they are augmented with narrative English prose.

1.3 Elementary Lexical Constructs

Pascal language lexical units – identifiers, basic symbols, and constants - are constructed from one or more (juxtaposed) elements of the alphabet described below.

1.3.1 Alphabet

SVS Pascal uses an extended form of the ASCII character set for all text related processing. ASCII is the American Standard Code for Information Interchange. There are 128 characters in the ASCII character set: 52 letters (upper case 'A' through 'Z', and lower case 'a' through 'z'), 10 digits, space (often called "blank"), 33 "control codes" (such as "carriage return" and "line feed"), and 32 graphic characters such as colon, equals sign, and so on. Pascal also allows an additional 128 values to be used as data values, for a total of 256 possible data values.

The Pascal compiler recognizes the following *alphabet* or *character set*:

<letter>	::=	'A' .. 'Z', 'a' .. 'z', and '_'
<digit>	::=	'0' .. '9'
<hex digit>	::=	<digit> | 'a' .. 'f' | 'A' .. 'F'
<ASCII graphic characters>	::=	! " # $ % & ' ( ) * = + - , . / < > ? [@ ^ | ~ ` { } ; :]

Note that the definition of <letter> above includes the underline character.

1.3.2 Pascal Identifiers

Pascal *identifiers* serve to denote constants, variables, procedures, and other language objects.

<identifier> ::= <letter> { <letter> | <digit> }

A Pascal identifier must start with a letter or an underline character. It can contain letters, digits, and the underline character. The underline is usually used to mark off spaces in the identifier to provide for readable and meaningful names. A Pascal identifier may be any length, but only the first 31 characters are significant to the compiler. Upper and lower case letters are all "folded" to a single case in the compiler, making them equivalent.

Examples of Identifiers

here_and_there	August_1979	Steve_and_Jeff
_X25	Tau_Epsilon_Xi	DragonsEgg
UPanddown	upandDOWN	upANDdown

The last three identifiers in the examples are equivalent because the compiler folds letters to a single case.

Examples of Invalid Identifiers

```
1st_character_must_be_a_letter
mustn't_have_odd_#$"[_characters_in_it
```

1.3.3 Numbers

Numbers are used to denote **integer**, **real**, and **double** data elements. Integers are assumed to be in the decimal number base, unless designated as a hexadecimal number.

```
<unsigned integer>  ::= <digit> {<digit>}

<unsigned real>     ::=
                        <unsigned integer>.<unsigned integer>
                    |   <unsigned integer>.<unsigned integer>E<scale factor>
                    |   <unsigned integer>E<scale factor>
                    |   <unsigned integer>.<unsigned integer>D<scale factor>
                    |   <unsigned integer>D<scale factor>

<unsigned number> ::= <unsigned integer> | <unsigned real>

<scale factor>      ::= <unsigned integer> | <sign><unsigned integer>

<sign>              ::= + | -

<hex number>        ::= $<hex digit> {<hex digit>}
```

Hexadecimal numbers are considered unsigned, unless they are explicitly written as 32-bit values with the most significant bit a one. For instance, the value $ffff is 65535 and not −1. The value $ffffffff is a negative number.

integer numbers are represented internally in the two's complement notation. As a consequence, there is one more negative integer than there are positive integers.

Values of type **double** are designated by a letter **D** preceding the exponent part of the number.

Examples of Valid Numbers

```
666
+99   -457      { unsigned decimal integer }
$3e8            { a hexadecimal number     }
0.0             { the real number zero     }
3.14159
1.23D10         { a double number          }
```

Examples of Invalid Numbers

```
5.      { should be a digit after the point    }
 .618   { should be a digit before the point   }
5.E10   { should be a digit after the point    }
2FC9    { Invalid decimal number               }
F034    { An identifier, not a hex number      }
```

1.3.4 Pascal Strings

Sequences of characters enclosed in apostrophes are called **strings**. Strings of one character are constants of type **char**. A string of "n" characters, where "n" is greater than one, is an ambiguous constant that is either a **string** value, or is a value of the type **packed array** [1 .. n] **of char**. The exact type of such a string constant is determined from the context in which it appears.

A string constant which is just simply two juxtaposed apostrophes '' represents a variable **string** constant of length zero.

SVS Pascal provides for entering any character value into a string by coding its two-digit hexadecimal value preceded by a reverse slash \. This means that non-printing characters such as "BEL" and "ETX" may be entered into a string. A \ sign followed by a non-hexadecimal digit is simply that character. Thus '\Y' is equivalent to 'Y', '\\' represents '\' and '\3X' represents '\03X'. This last case is interesting in that leading zeros are implicit in the hexadecimal number if there is only one hexadecimal digit followed by a non-hexadecimal digit.

An apostrophe in a string is represented by two juxtaposed apostrophes. The rules for reverse slash character representations above means that an apostrophe can also be represented by the string '\'', or by the string '\27'.

```
<string>           ::= '<character> {<character>}'
<character value>  ::= <two digit hexadecimal number>
```

Examples of Strings

```
'This is a string constant'

'This string has an embedded '' apostrophe'

'here is how to get a \07 bell character in a string'

'to get a back slash, just type \\'
```

1.3.5 Pascal Labels

A *label* is used to mark statements as the potential target of a **goto** statement.

Pascal labels are unsigned **integer** constants in the range 0 .. 9999.

<label> ::= <unsigned integer>

1.3.6 Basic Symbols

Pascal has a set of "basic symbols" which the compiler uses for specific purposes in the language. These basic symbols include selected identifiers (reserved words), graphic characters, and pairs of graphic characters. These basic symbols are used as keywords, operators, delimiters and separators. Such symbols are introduced throughout the body of this manual.

Note that user-defined identifiers may not be the same as any Pascal reserved word.

Identifiers (reserved words) used as basic symbols are shown in this manual in bold faced typefont. For example, **procedure**, **else**, and **type** are Pascal reserved words.

There are two lists of basic symbols shown below. One is a list of Pascal reserved words and the other is a list of the special graphic symbols that Pascal uses.

Pascal Reserved Words

and	**end**	**label**	**program**	**until**
array	**file**	**mod**	**record**	**uses**
begin	**for**	**nil**	**repeat**	**var**
case	**function**	**not**	**set**	**while**
const	**goto**	**of**	**string**	**with**
div	**if**	**or**	**then**	
do	**implementation**	**otherwise**	**to**	
downto	**in**	**packed**	**type**	
else	**interface**	**procedure**	**unit**	

Pascal Special Symbols

+	Adding Operator.
−	Subtracting Operator.
*	Multiplying Operator.
/	Division Operator (for **real** and **double** data types).
:=	Assignment Operator.
.	Terminates a Pascal Compilation Unit; Separates integer from fraction in a **real** or **double** number; Indicates reference to a field of a record.
,	Separates items in lists.
;	Statement and Declaration Separator.
:	after case and statement labels; variable and parameter descriptions.
'	string delimiter.
=	Relational equality operator; Used in constant and type definition.
<>	Relational operator for inequality.
<	Relational operator for "less than".
<=	Relational operator for "less than or equal to".
>=	Relational operator for "greater than or equal to".
>	Relational operator for "greater than".
(and)	encloses lists of elements; encloses parts of expressions that are to be considered indivisible factors.
[and]	encloses **array** subscripts and lists of **set** elements.
{ and }	comment delimiters.
(* and *)	are an alternative form of comment delimiters.
^	pointer dereference operator.
..	indicates a range of elements.

1.3.7 Conventions for Spaces

Spaces (also called blanks) are used to separate lexical items. Identifiers, reserved words and constants must not abut each other, neither may they contain embedded spaces. Multiple-character basic symbols such as <= must not contain embedded spaces.

Other than that, spaces may be used freely (to improve program readability for instance). They have no effect, outside of character and string constants, where a space represents itself.

1.3.8 Comments

Comments in Pascal may appear anywhere that a space may appear, and in fact, serve the same purpose as do spaces. But note that a comment within a string constant is part of the string constant and is not really a comment. Pascal comments are enclosed between braces {...} or between the characters (* and *).

```
<comment>  ::=  { <any printable characters except "}"> }
                | (* <any printable characters except "*)" *)
```

In the description above, the braces enclosing the comment are the comment delimiters, not metalanguage symbols.

For historical reasons, Pascal accepts two forms of comment delimiters. The open and close braces { } can be used where the character set provides such. Most modern computer systems and terminals accommodate those characters. Those systems which do not accommodate the full ASCII character set can use the alternative forms of (* and *) to delimit comments.

Comments that start with one kind of opening delimiter must end with the corresponding closing delimiter. For example:

```
{  this Pascal comment is enclosed in braces  }

(*  this comment uses the alternative delimiter  *)

{  this Pascal comment would go on for ever  because *)
(*  does not close the comment.  For that we need a closing brace  }
```

Pascal comments can span multiple lines, thus providing a "block comment" capability.

Chapter 2 – Defining Data Types

One of Pascal's major attractions is the ease with which users can describe and manipulate data. An important aspect of structured programming technology is the ability to structure data as well as control statements. This is provided in Pascal through the notion of a data type.

A *type* defines a collection of values that a variable, constant or expression may take on. A type has an associated size, but of itself reserves no storage space. Storage is only reserved when a variable is declared as an instance of that type. Although Pascal data types can be quite complex, they are ultimately composed of simple unstructured components. An example is the predefined type **integer**. Its size is two bytes (16 bits). The set of values it contains is −32768, ..., −1, 0, 1..., 32767.

In addition to having a size and a set of values, a type has a collection of operations in which values of that type can participate.

Pascal provides a number of predefined types (some of which were described in Chapter 1), as well as the means for users to define their own types. Section 2.2 of this chapter describes all predefined Pascal types.

Type constructors are the means by which users can define their own types. Structured type constructors facilitate the definition of new and larger types based upon other existing types as components.

2.1 Defining Constants

A literal constant is a value that denotes itself – its value is manifest from its appearance. The integer 1776 and the string 'Manila' are literal constants. A constant definition introduces an identifier that is a synonym for a constant. Using the identifier is equivalent to using the associated literal constant. Whereas the string "3.14159" is a literal constant, an identifier called "Pi" could be defined which is a synonym for the number. The identifier is then

known as a constant identifier, or just a constant.

```
<constant identifier> ::= <identifier>

<constant>        ::= <unsigned number>
                    | <sign> <unsigned number>
                    | <constant identifier>
                    | <sign> <constant identifier>
                    | <string>

<constant definition> ::= <identifier> = <constant>;
```

The definition above means that a constant may be defined to be another constant, but prohibits constant expressions.

2.1.1 Predefined Constants

Pascal provides three constants that are automatically declared as part of the language. The three constants are:

true Represents the truth value for a **Boolean** type.

false Represents the falsity value for a **Boolean** type.

maxint An **integer** constant representing the largest integer that Pascal can store. Maxint is currently defined as 32767.

Examples of Constant Definitions

```
Liters_per_bottle = 0.750;          { standard bottle is 750 ml }
Bottles_per_Case = 12;              { standard case             }
first_vowel = 'a';                  { a char constant           }
Winery = 'Chateau Montelena';       { a string constant         }
Carriage_Return = '\0d';            { carriage return character }
```

2.2 Standard Types

SVS Pascal has eight predefined types available:

integer integer type represents an implementation defined subset of the integers. It is equivalent to a subrange defined by a type definition that looks like:

integer = −32768 .. 32767

The **integer** data type therefore occupies 16 bits of data

storage.

longint is a long integer type. It is equivalent to a subrange defined by a type definition that looks like:

longint = −2147483648 .. 2147483647

The **longint** data type therefore occupies 32 bits of data storage.

real real type is a subset of the continuum of real numbers. Reals are represented in the "floating point" format which consists of a fractional part (a mantissa) and an exponent. The range of real numbers is approximately −3.4E38 .. +3.4E38, with a precision of approximately seven decimal places. In addition, the **real** data type can take on "extreme values", such as plus infinity, minus infinity, and "Not a Number" (abbreviated NaN), which arise from overflow and division by zero. There is a detailed discussion of extreme values in Appendix E − "Data Representations".

double double type is a double precision form of the **real** data type described above, and is a subset of the continuum of real numbers. Double numbers are represented in the "floating point" format which consists of a fractional part (a mantissa) and an exponent. The range of **double** numbers is approximately −1.8D308 .. +1.8D308, with a precision of approximately 15 decimal places. In addition, the **double** data type can take on "extreme values", such as plus infinity, minus infinity, and "Not a Number" (abbreviated NaN), which arise from overflow and division by zero. There is a detailed discussion of extreme values in Appendix E −"Data Representations".

Boolean **Boolean** type represents the ordered set of "truth values" whose constant denotations are **false** and **true**. **Boolean** is conceptually equivalent to an ordinal type specified by a type definition that looks like:

Boolean = (false, true)

char **char**acter type defines the set of 256 values of the ASCII character set, and is equivalent to the subrange defined by a type definition that looks like:

char = '\0' .. '\255'

An unpacked **char** data item occupies one word or 16 bits of data storage. A packed **char** data item occupies one byte or 8 bits of data storage.

text is equivalent to a **packed file of char**.

interactive is a file type the same as that of **text**, except that the standard procedures READLN and WRITELN treat the end-of-line in a way that is more suitable for interactive (terminal) devices.

2.3 Defining Data Types

Pascal data types (or just *types* for short), are used to define sets of values that Pascal variables may assume and in many cases, a notation for referencing such variables. Pascal provides a small number of predefined types, reserved identifiers for these types, and a notation for defining new types in terms of existing types.

Type declarations introduce new (user defined) types, and identifiers for those newly-declared types.

<type spec> ::= <type identifier> = <Pascal type>;

Type declarations can be used for purposes of brevity, clarity and accuracy. Once declared, a type may be referred to elsewhere in the program by its declared type-identifier.

2.4 Simple Types

Simple types are those that have neither structure nor components. The simple types are as follows:

```
<simple type> ::= <scalar type>
                | <standard type>
                | <subrange type>
                | <type identifier>
```

2.4.1 Scalar Types

A *scalar type* defines a well-ordered set of values by enumerating the identifiers that denote those values. A scalar type is also known as an *enumerated* type or an *ordinal* type. An ordinal type is represented by the ordered set of integers 0, 1, 2, 3, ..., with the first identifier being 0, up to the last identifier which is "n"−1, where "n" is the number of identifiers in the list.

<scalar type> ::= (<identifier> {,<identifier>})

Examples of Scalar Type Definitions

```
salad_greens  = (Spinach, Lettuce, Coriander,
                 Escarole, Watercress);
bottle_sizes  = (Fillette, Bottle, Magnum, Marie_Jeanne,
                 Jeroboam, Imperial);
mealtimes     = (Breakfast, Elevenses, Lunch,
                 AfternoonTea, Dinner);
```

2.4.2 Subrange Types

A *subrange type* represents a subrange of values of another scalar type. It is defined by a lower and an upper bound. The lower bound must not be greater than the upper bound, and both bounds must be of identical scalar types.

Values from a subrange and values from its parent range (or another subrange of its parent range) can be assigned to each other and can enter into the operations of assignment, comparison, and other binary operations.

```
<subrange type> ::=
        <subrange type identifier> | <lower> .. <upper>

    <lower>  ::= <signed scalar constant>
    <upper>  ::= <signed scalar constant>
```

Examples of Subrange Type Definitions

```
small_integer      = 0 .. 15;
days_in_year       = 1 .. 366;
positive_integer   = 0 .. 32767;
lower_case_letters = 'a' .. 'z';
colors             = (red, orange, yellow, green, blue);
hot_colors         = red .. yellow;
cold_colors        = green .. blue;
hues               = red .. blue;
days               = (Saturday, Sunday, Monday, T, W, T, Friday);
weekdays           = Monday .. Friday;
weekends           = Saturday .. Sunday;
```

2.5 Structured Types

Structured types represent collections of objects. They are defined by describing their element types and indicating a *structuring method.* These differ in the accessing mechanisms and in the notation used to select elements from

the collection.

Pascal makes available five structuring methods: **array**, **string**, **set**, **record** and **file**. Each type is described in the subsections to follow.

A structured type may be given the **packed** storage attribute. This "advises" the compiler that the structure is to use data storage economically, by packing the components of the structure densely. Packing is often achieved at a cost of larger code size and slower execution speed. Furthermore, a component of a **packed** variable can not be passed as a **var** parameter to a procedure or function (this restriction applies to components of **packed array of char**). A full discussion on how components are packed can be found in Appendix E – "Data Representations".

```
<structured type>          ::=<unpacked structured type>
                             | packed <unpacked structured type>

<unpacked structured type> ::=<array type>
                             |   <string type>
                             |     <record type>
                             |       <set type>
                             |         <file type>
```

2.5.1 Array Types

An **array** type is a structure consisting of a fixed number of components, all of the same type (called the *component type*). Array elements are designated by indexes, which are values belonging to the so-called *index type*. The array type-definition specifies the component type as well as the index type.

```
<array type>  ::=  array [<index list>] of <type>
<index list>  ::=  <simple type> {, <simple type>}
```

If "n" index types are specified, the array is an "n" dimensional array. Note that the above definition for an **array** type means that there are two alternative ways of specifying an array. By definition, a component of an array can be another array type. Thus a three dimensional array could be specified as follows:

```
blivet  =  array [1..10, 11..20, 21..30] of blimps;

widget  =  array [1..10] of array [11..20] of
           array [21..30] of blimps;
```

The alternative forms of specifying array types are equivalent. The first form can be thought of as a shorthand notation for the second form. There is a similar choice of notations when specifying the index elements for accessing an array component.

When the index type is a subrange of the type **integer**, the type:

packed array [1 .. n] of char

is a special case. Objects of this type up to a maximum length of 255 characters can be compared as single entities, whereas arrays of other data types must be compared element by element. A literal string constant can be assigned to a **packed array of char**, providing that the lengths are the same. The type of a literal string of length 'n', where 'n' is greater than 1 is compatible with the type:

packed array [1 .. n] of char

Examples of Array Type Definitions

```
rows = 1 .. 3;
columns = 1 .. 4;

bottle_quantities = array [bottle_sizes] of integer;

standard_case = packed array [rows]
                              of array [columns]
                                          of bottles;

token = packed array [1 .. 100] of char;
```

2.5.2 String Types

SVS Pascal has a structured type constructor called **string**. A **string** variable has a maximum length (called its static length) which is determined when the string is defined. A **string** variable also has a dynamic length which can vary over the range 0 through its static length during execution of a program. The standard function LENGTH can be used to determine the string's dynamic length. The maximum static length of a **string** variable is 255 characters.

Strings can be manipulated by standard Pascal syntax, or by using string handling intrinsics, described in Chapter 8 — "Standard Procedures and Functions".

<string type> ::= **string[**<static length>**]**

<static length> ::= integer constant in the range 1 .. 255

A **string** constant which is '' (two juxtaposed apostrophes) represents a null or zero-length string.

Example of String Type Definition

```
manila = string[100];
punched_card = string[80];
```

2.5.3 Record Types

A *record type* is a structure consisting of a fixed number of components that may be of different types. For each component, or *field* of the record, the definition specifies both a type and an identifier used to reference the field. The scope of these *field identifiers* is the definition of the record itself. This means that the same field identifier may appear in more than one record. A field identifier is also accessible within a field designator when referring to a variable of this record type.

Record components which are themselves records do not inherit the packing attribute of the containing record. Each component which is a record has independent packing attributes.

A *variant record* caters to the need for a record composed of a portion which is always the same, plus one or more *variants* whose layouts differ between different instances of the record. The specific variant that is selected in any given instance is determined by an optional *tag field.* Such a structure is called a *variant record* or a discriminated union. The tag field is often called a discriminant. The tag field's value indicates which variant the record assumes at a given time. Each variant structure is identified via a case label which is a constant of the tag field's type. Referencing a field of a variant that is inconsistent with the tag fields's value is a serious programming error.

```
<record type> ::= record <field list> end;

<field list> ::= <fixed part>
               | <fixed part> ; <variant part>
               | <variant part>
<fixed part> ::= <record section> {; <record section>}
<record section> ::= <field identifier list> : <type>
<field identifier list> ::= <field identifier> {,<field identifier>}

<variant part> ::=
             case {<tag field>} <type identifier> of <variant list>
<variant list> ::= <variant> {; <variant>}
<variant> ::= <case label list> : (<field list>)
<case label list> ::= <case label> {, <case label>}
<case label> ::= <constant>
<tag field> ::= <identifier>:
```

Note that the <tag field> is optional in a variant record definition.

Examples of Record Type Definitions

```
{ the example to follow illustrates an      }
{ ordinary record called ComplexNumber,     }
{ which contains two fields, namely the     }
{ real part and the imaginary part.         }

ComplexNumber = record
                    RealPart: real;
                    Imaginary: real;
                end;

{ The example below illustrates a variant   }
{ record type which has different sections  }
{ that are accessed depending on the tags.  }
{ First we define an enumerated type which  }
{ is used as the variant case selector.     }
shapes = (rectangle, triangle, circle, polygon);

angle = -180 .. +180;

PositionRec = record
                  x_position: real;
                  y_position: real;
                  case whatshape: shapes of
                   rectangle: (base: real;
                               height: real);
                   triangle: (base: real;
                               height: real;
                               skew: angle);
                   circle: (radius: real);
                   polygon: (SideCount: integer;
                               radius: real);
              end;
```

2.5.4 Set Types

A **set** type definition serves to define the base type that the set is to use in future manipulations. Sets are limited to 2032 elements. The range of the set elements must be within the range 0 .. 2031.

<set type> ::= **set of** <simple type>

Examples of Set Type Definitions

```
salad_base  =  set of salad_greens;
dressings   =  set of salad_dressings;
lower_case  =  set of 'a' .. 'z';
```

2.5.5 File Types

A *file type* defines a sequence of elements. A file is usually associated with external storage devices or communication devices. SVS Pascal supports the standard Pascal typed files, untyped files and an **interactive** file type more suitable for terminals.

When a file variable "f" with components of type "T" is declared, there is an additional implied declaration of a so called *buffer variable* or "window", also of type "T". This window is referenced by the notation f^ where "f" is the file variable. This window is used in conjunction with the GET and PUT procedures (see Chapter 6 – "Input and Output") and serves to append components to the file when writing, and to access the components when reading from the file.

<file type> ::= **file of** <type>
| **file**

SVS Pascal supports *untyped* files. An untyped file can be considered to not have a window variable. Such files must be accessed using the BLOCKREAD and BLOCKWRITE functions described in Chapter 6 – "Input and Output".

A file of the pre-defined type **text** can be considered to be defined by a type definition of the form:

text = packed file of char;

Such a file is special in that the range of its components (characters) are extended to include an end-of-line marker. Such a file can then be conveniently structured into lines. The EOLN predicate described in Chapter 8 – "Standard Procedures and Functions", covers how the end-of-line is detected.

SVS Pascal also supports an **interactive** file type which display different behavior in the way that the READ, READLN and RESET intrinsics work.

The differences are covered in Chapter 6 – "Input and Output". An **interactive** file is more suitable for use with interactive terminals.

Examples of File Type Definitions

```
block_access  = file;
numbers       = file of integer;
Capping_Line  = file of bottles;
Terminal      = interactive;
legible_file  = text;
```

2.6 Pointer Types

Explicitly declared variables are accessible by reference to the identifier used to declare them. Such variables are accessible during the activation (scope) of the procedure in which they are declared. These variables are called *static*, that is, lexically static.

Variables may also be created dynamically, in other words, with no correlation to the program structure. These dynamic variables are created via the procedure NEW. Since such variables do not have an associated name, they are accessed via a *pointer* value which is generated when the variable is allocated. A pointer type is therefore a value which points to a variable of a specific type.

There is a universal pointer value called **nil**, which belongs to any pointer type. It represents a pointer which points to no element.

<pointer type> ::= ^<type identifier>

Examples of Pointer Type Definitions

```
blackboard = record
   long_side : integer;
  short_side : integer;
end;

cue = ^blackboard;

TwoWay = record
            next: ^TwoWay;
            previous: ^TwoWay;
            stuff: array[0 .. 10] of integer;
         end;

SymTree = record
            name: string[31];
            LeftNode: ^SymTree;
            RightNode: ^SymTree;
         end;
```

2.7 Type Identity and Assignment Compatibility

Pascal has strict type checking such that objects of one type cannot be combined in operations with objects of a different type. There are two major concepts to be described here, namely *identical types* and *assignment compatible types.*

2.7.1 Identical Types

Two types, T1 and T2 are considered identical under the following conditions:

- T1 and T2 are the same type.
- T1 is declared as synonymous with another type T3, where T2 and T3 are identical.

Examples of Type Identity

```
type_x  = integer;
type_y  = integer;
type_1  = set of char;
type_2  = set of char;
id_type = type_1;
```

In the above example, the types "type_x" and "type_y" are identical, because they are defined to be the same type, **integer**. The types "type_1" and "type_2" are not identical, since they occur in different type definitions. The types "type_1" and "id_type" are identical however, because "id_type" is defined to be the same as "type_1".

2.7.2 Assignment Compatible Types

A value of type T1 is considered to be assignment compatible with a variable of type T2 if any of the following conditions are true:

- T1 and T2 are identical *and* do not contain a file as a component.
- T1 is a subrange of T2, or
- T2 is a subrange of T1, or
- T1 and T2 are subranges of identical types.
- T1 is assignment compatible with **integer** and T2 is **real** or **double**.
- T1 and T2 are both variable **string** types.
- T1 and T2 are sets of elements of types T3 and T4, and T3 is assignment compatible to T4.

Chapter 3 – Variables

This chapter covers two topics. First there is a discussion of how Pascal *variables* are declared in terms of the data types described in the previous chapter. Then there is a description of the way that variables of different types are accessed or referenced.

3.1 Declaring Variables

A *variable* has a type and a storage area in memory. At any given time, a variable takes on one value out of the collection of values that define its type. A variable is initially undefined, and remains so until it is initialized by an explicit assignment.

All variables in a Pascal program must be declared explicitly and prior to their use.

Variable declarations consist of a list of identifiers that represent the variables, followed by the type of the variable.

```
<variable declaration> ::=
        <identifier> {,<identifier>}: <data type>;
```

Examples of Declaring Variables

```
Impedance: ComplexNumber;  { a record variable    }
ChainHead: TwoWay;         { another record       }
TreeTop: SymTree;          { and another          }
First, Middle, Last: integer;  { plain integers   }
ValueFile: Numbers;        { a file variable      }
CurChar: char;             { a character variable }
Omega: real;               { a real variable      }
```

3.2 Predeclared Variables

SVS Pascal has five pre-declared variables. These are:

input, **output**, and **stderr** default files associated with the standard input, the standard output, and the standard error output file, respectively. On those operating systems which do not have a standard error output file, the file **stderr** is directed to the same place as the **output** file.

argc and **argv** are variables which provide access to the command line that invoked the current Pascal program.

These pre-declared variables are covered in detail in Chapter 7 — "Program Structure".

3.3 Establishing Variables

Establishing a variable is a process that involves:

1. determination of the variable's type.
2. allocation of storage for the values that the variable takes on.

Explicitly declared variables are automatically established on each entry to the procedure or function block in which they are declared. "Global" variables (declared in the outermost block) are established once and only once.

Formal parameters of procedures or functions are automatically established on each activation of that procedure or function.

So-called "dynamic" variables are explicitly established by storage management operations (for type determination and storage allocation), and by assignment operations (for initialization).

3.4 Lifetimes of Variables

The lifetime of a local variable is that of the block in which it is declared. Allocation occurs on each entry to that block, and de-allocation occurs on each exit from that block.

3.4.1 Global Variables

Global variables are those variables declared in the outermost block (in the **program** block). The lifetime of such global variables is the lifetime of the entire program.

3.4.2 Lifetime of Formal Parameters

The lifetime of a formal parameter is the lifetime of the procedure or function which that formal parameter is a part of. The formal parameter becomes established upon each entry to the procedure or function, and becomes undefined upon exit from the procedure or function.

3.4.3 Lifetime of Dynamic Variables

Dynamic variables are established (but not initialized) by an explicit allocation operation (such as NEW). Dynamic variables become undefined when they are explicitly freed, or when no pointer variable points to them. Note that generally a pointer value has a finite lifetime which may be different from that of the pointer variable that can point to it. Local variables belonging to procedures and functions, cease to exist on exit from the block in which they were declared. Dynamic variables, on the other hand, cease to exist when they are explicitly freed or when no pointer variable points to them. Attempts to reference non-existent variables beyond their lifetimes is a programming error, usually with undesirable results from the programmer's viewpoint.

3.5 Referencing or Accessing Variables

The method by which a variable or a component of a variable is accessed differs depending on the structuring method used in the type definition for that variable. There are three basic access methods:

1. An *entire variable* is a variable of a simple type (no structure). An entire variable is referenced simply by giving its name.
2. A *component variable* is a variable of **array**, **record** or **file** type. The access methods are explained below.
3. A *referenced variable* is accessed through a pointer.

```
<variable> ::= <entire variable>
             |   <component variable>
             |     <referenced variable>
```

3.5.1 Entire Variables

An entire variable is denoted by its identifier. Since an entire variable has no structure, its identifier alone is enough to reference it.

```
<entire variable> ::= <variable identifier>
```

Examples of Entire Variable References

ChickenTeeth
GiddyGoatHorns
First

3.5.2 Component Variables

A component of a variable is denoted by the variable followed by some selector that specifies the component. The form of the selector depends on the structuring method used to access the variable.

```
<component variable> ::= <indexed variable>
                       | <field designator>
                       | <file buffer>
```

3.5.2.1 Referencing Indexed Variables

A component of an "n"-dimensional array variable is denoted by the variable followed by "n" index expressions. An entire array (which can be a component) of an array can be denoted by giving "n"-1 index expressions. In such a case, the entire last dimension of the array is indicated. This occurs when an entire array or an entire subarray is passed as an actual parameter to a procedure or function.

```
<indexed variable>  ::= <array variable> <subscript list>

<subscript list>    ::= [ <expression> {,<expression>} ]
                      | [ <expression> ] {[ <expression> ]
```

The {,<expression>} in the definition above implies that there are as many expressions in the subscript list as there are dimensions in the array variable. Just as in defining an **array** type, there are two alternative methods for referencing an array variable. Either the subscripts can be listed, separated by commas, inside the brackets, or there can be a list of bracketed subscript expressions.

The index expression types must correspond with the index types declared in the array type definition.

Examples of Array Variable References

ladder[top]

stairs[flight][step]

Footing[Left, Center, Right]

3.5.2.2 Referencing Strings

String variables can be referenced as single entities (when the entire string is being operated upon) or single characters from a string can be referenced just like a **packed array of char**. Values can be assigned to string variables using assignment statements, string intrinsics or the READ or READLN procedure. String indexing is based from one (1) so that the expression on the string "s":

s[LENGTH(s)]

correctly yields the last character in the string. The dynamic length of the string may be addressed as the zeroth element of the string. Thus the statement 's[0] := chr(3)' would set the dynamic length of the string to 3. The length must never be set to a value greater than the maximum declared for that string! It is an error to reference a string "s" with an index less than zero or greater than LENGTH(s).

3.5.2.3 Referencing Fields of Records

A component of a record variable is denoted by the record variable followed by the component's field identifier. The field identifiers are separated by periods.

<field designator> ::= <record variable>.<field identifier>

It is an error zero (which is not flagged by the Pascal system) to reference a field of a variant record that is inconsistent with the tag field for that variant.

Examples of Accessing Fields in Record Variables

```
        { The first example is a simple field reference }
impedance.RealPart

        { The second example illustrates a reference
          to a field of an array of records          }
bottles[BurgundyType].Loire

        { The third example illustrates a
          deeply nested field reference      }
King_Caractacus.Court.Ladies.Faces.Noses
```

3.5.2.4 Referencing File Buffers

At any time, only the one component determined by the current file position (read/write head) is directly accessible. This component is called the "current file component", and is represented by the file's buffer variable.

<file buffer> ::= <file variable>^

<file variable> ::= <variable>

3.5.3 Pointer Referenced Variables

<referenced variable> ::= <pointer variable>^

<pointer variable> ::= <variable>

If "p" is a variable which is a pointer to type "T", "p" means the pointer variable and its pointer value, whereas "p^" means the variable of type "T" that "p" references.

Examples of Pointer Reference

```
TreeTop.LeftNode^  { Left Node in the tree variable }
cue^.longside      { gets Long Side of Blackboard   }
```

Chapter 4 – Expressions

An expression is a construct which defines the rules of computation for creating a value by performing operations (specified by operators) on operands (specified by variables, constants, and function references). These newly-created values can then be used in assignment statements or can be used (in conditional expressions) to control subsequent program actions.

```
<unsigned constant> ::= <unsigned number
                      | <string>
                      |   <constant identifier>
                      |     nil

        <factor>    ::= <variable>
                      | <unsigned constant>
                      |   <function designator>
                      |     <set constructor>
                      |       (<expression>)
                      |         not <factor>

  <set constructor> ::= [ <element> {,<element>} ]
       <element>    ::= <expression>
                      | <expression> .. <expression>

          <term>    ::= <factor>
                      | <term> <multiplying operator> <factor>

    <simple expr>   ::= <term>
                      | <simple expr> <adding operator> <term>
                      |   <adding operator> <term>

    <expression>    ::=
                        <simple expr>
                      | <simple expr> <relational operator> <simple expr>
```

4.1 Operators in Expressions

Operators perform operations on a value or a pair of values to produce a new value. Most operators are defined only on basic types, though some range, as well as the result, of the defined operators.

With the exception of the @ operator, an operation on a variable or field which has an undefined value, produces an undefined result.

4.2 Address Evaluation Operator

The @ operator generates the address of a variable, user procedure or user function. The type of the resulting expression is the same as the type of the value **nil**. Thus an address can be assigned to any pointer variable.

The precedence of the @ operator is above that of all other operators, but below that of array indexing and record field referencing. It can be applied to unpacked fields of records and unpacked array elements and to the dynamic variables pointed to by a pointer. It cannot be applied to components of any packed structure.

Examples of the @ Operator

@Uncle_Bill	generates the address of a variable named "Uncle_Bill".
@TypeWheel[tilde]	generates the address of the "tilde"th element of the array "TypeWheel".

4.3 NOT Operator

The **not** operator applies to factors of type **Boolean** or **integer**.

When applied to type **Boolean**, the meaning is negation. That is, **not true = false**, and **not false = true**.

When applied to type **integer**, the **not** operator negates all the bits in the value. That is, it performs a one's complement negation of each bit in the operand. The result of applying the **not** operator to a value of type integer is type **integer**.

4.4 Multiplying Operators

The multiplying operators have the next highest precedence after the **not** operator.

<multiplying operator> ::= * | \ | **div** | **mod** | **and**

The following table shows the multiplying operators, the permissible types of their operands, and the result types. Operands of the *

(multiplication) and / (division) operators can be mixed **integer**, **real**, and **double** data types.

If both operands of the * operator are of type **integer**, the result is of type **integer**.

If either operand is of type **double**, the other operand is converted to type **double**, and the result is of type **double**. Otherwise, if either operand is of type **real**, the result is of type **real**. The result of the / operator is either **real**, or in the case when one or both operands are of type **double**, the result is of type **double**.

Operator	Operation	Operands	Result
*	multiplication set intersection	**real**, **double**, or **integer** any set type T	**real**, **double**, or **integer** T
/	division	**real**, **double**, or **integer**	**real** **double**
div	division with truncation	**integer**	**integer**
mod	modulus	**integer**	**integer**
and	logical and bitwise and	**Boolean** **integer**	**Boolean** **integer**

The **div** operator applies to values of type **integer** only and represents truncating division. **div** always truncates towards zero. It is an error to divide by zero. If the signs of the operands are the same, the result is positive; if the signs are different, the result is negative.

The **mod** operator defines the modulus operation between two values of type integer. It is an error if the right operand of **mod** is zero. The interpretation of **mod** is:

a **mod** b = a − (a **div** b) * b

When applied to operands of type **Boolean**, the **and** operator produces a result of type **Boolean** as one might expect. When applied to operands of type **integer** however, the **and** operator performs a bitwise logical and on the operands and produces a result of type **integer**.

4.5 Adding Operators

The adding operators have the next highest precedence after the multiplying operators.

<adding operator> ::= + | − | **or**

The following table shows the adding operators, their permissible operand types, and the result types. Operands of the + (addition) and − subtraction operators can be mixed **integer**, **real**, and **double** data data types.

If both operands of the + or − operator are of type **integer**, the result is of type **integer**.

If either operand is of type **double**, the other operand is converted to type **double**, and the result is also of type **double**. Otherwise, if either operand is of type **real**, the result is also of type **real**.

operator	operation	operand types	result type
+	addition set union	**real**, **double**, or **integer** any **set** type T	**real**, **double**, or **integer** T
−	subtraction set difference	**real**, **double**, or **integer** any **set** type T	**real**, **double**, or **integer** T
or	logical or bitwise or	**Boolean** **integer**	**Boolean** **integer**

When applied to operands of type **Boolean**, the **or** operator produces a result of type **Boolean** as one might expect. When applied to operands of type **integer** however, the **or** operator performs a bitwise logical or on the operands and produces a result of type **integer**.

4.6 Sign Operators

The "+" and "−" signs can be used as unary operators. They apply to **integer**, **real**, and **double** data types only. Applying a unary operator to a data type produces a result which is the same data type as that of the operand.

<sign operator> ::= + | −

The table below shows the sign operators, their permissible operand types and their result types.

operator	operation	operand types	result type
+	identity	**real, double,** or **integer**	**real, double,** or **integer**
−	negation	**real, double,** or **integer**	**real, double,** or **integer**

4.7 Relational Operators

The following table shows the relational operators, their permissible operand types, and the result type.

operator	operand types	result type
= <>	any scalar or subrange type **set** type pointer type **packed array of char** **string**	**Boolean**
<= >=	any scalar or subrange type **set** type **packed array of char** **string**	**Boolean**
< >	any scalar or subrange type **packed array of char** **string**	**Boolean**
in	any scalar or subrange type and its **set** type respectively.	**Boolean**

Note that all scalar types define ordered sets of values.

4.7.1 Comparison of Scalars

All six relational operators (<, <=, >, >=, = and <>) are defined between operands of the same scalar type.

For operands of type **integer**, **real**, or **double**, the operators have their usual meaning. Operands of **integer**, **real**, and **double** data types are

considered to form a hierarchy, with the **integer** data type at the bottom of the pecking order, the **double** data type at the top, and the **real** data type in the middle. If the operands are of different numeric types, the lower type of operand is converted (or promoted) to the type of the other operand prior to the comparison. For example, in the expression:

integer type < **double** type

the **integer** operand is converted to **double** before the comparison is made.

For operands of type **Boolean** the relation **false** < **true** defines the ordering.

For operands of type **char** the relation "a" *op* "b" holds if and only if the relation ORD(a) *op* ORD(b), holds, where **op** denotes any of the six comparison operators and **ord** is the mapping function from type **char** to type **integer** defined by the ASCII collating sequence.

For operands of any ordinal type "T", "a" = "b" if and only if, "a" and "b" are the same value; "a < b" if and only if, "a" precedes "b" in the ordered list of values that define "T".

4.7.2 Comparison of Booleans

If "p" and "q" are **Boolean** expressions, "p = q" means equivalence, and "p < = q" means implication of "q" by "p".

4.7.3 Direct Pointer Comparison

Two direct pointers can be compared if they are pointers to identical types. To compare pointers of differing types, take their ORD. (See Chapter 8 — "Standard Procedures and Functions").

Pointers may be compared for equality or inequality only.

Two pointers with the value **nil** are always equal.

4.7.4 String Comparison

All six relational operators may be applied to **string** operands. The relational operators compare both **packed array of char** and **string** values.

In the case of a **packed array of char**, both operands must be the same size. The maximum length of string comparison of values of **packed array of char** is 255 characters. That is, a variable whose declaration is like:

```
var
    strtype: packed array [1 .. 255] of char;
```

is the largest string variable that can be compared in one operation.

In the case of **string** comparison, the operands may be of different sizes. If the operands are of different sizes, trailing spaces are significant. That is, the string

```
'A'
```

compares less than the string

```
'A '
```

Comparison of **string** operands or **packed array of char** operands denotes alphabetical ordering according to the ASCII character set collating sequence.

Note that because a **string** data type is represented differently from a **packed array of char**, they cannot be compared with each other. On the other hand, a character string *constant* is of ambiguous type, and so a string constant can be compared either to a **string** operand or to a **packed array of char** operand, because the type of the string constant is converted to the type of the other operand in comparison operations.

4.7.5 Set Comparison

The relation "scalar_value" **in** "some_set" is **true** if the "scalar_value" is a member of the "some_set". The base type of the set must be the same as, or a subrange of, the type of the scalar.

The set operations = (identical to), and <> (different from), <= (is included in), and >= (includes) are defined between two set values of the same base type. For two sets "S1" and "S2" of the same base type:

S1 = S2	is true if all members of S1 are contained in S2, and all members of S2 are contained in S1.
S1 <> S2	is true when S1 = S2 is false.
S1 <= S2	is true if all members of S1 are also members of S2.
S1 >= S2	is true if all members of S2 are also members of S1.

4.7.6 Non-Comparable Types

Certain Pascal types cannot be compared. These include files, arrays, variant records, and records containing fields of non-comparable types. The exception to this rule is that **packed array of char** operands can be compared if they are the same size.

4.8 Out of Range Values

It is possible that expression evaluation can yield results which are outside of the range of values for a given data type. Expressions involving the

real and **double** data types can generate several different extreme values.

The extreme value of positive or negative infinity is a result either of overflow, or by dividing a non-zero value by 0.0.

Underflow generates a value of zero.

Dividing 0.0 by 0.0 generates a value of Not a Number (NaN).

Appendix E – "Data Representations" contains a description of the extreme values and their behavior in comparisons.

4.9 Order of Evaluation in Expressions

The rules of composition for expressions specify operator precedence according to five operator classes. The precedence is as follows:

1. the "address of" @ operator has the highest precedence.
2. then the **not** operator.
3. then the multiplying operators.
4. then the adding operators.
5. the lowest precedence is the relational operators.

Operators at the same precedence level are applied left to right, except where parentheses are used to override the normal order of evaluation. The order in which operators are applied is according to the rules above. The precise order of operand evaluation is undefined. Some operands may not be evaluated at all, if the value of the expression can be determined without the value of that particular operand.

4.10 Compile Time Constant Expressions

The Pascal compiler evaluates certain types of **integer** and **Boolean** constant expressions at compile time. **integer** expressions consisting of constant expression operands and the following operators are folded into constant expressions:

Binary Operators = <> + – *
Unary Operators –

Boolean expressions consisting of constant expression operands and the following operators are folded into constant expressions:

Binary Operators = <> **and or**
Unary Operators **not**

4.10.1 Dead Code Elimination

The Pascal compiler recognizes code of the form:

```
if FALSE then
    statement_1
else
    statement_2
```

and generates code for statement_2 only. Similarly, if the **Boolean** expression is TRUE, only statement_1 is generated. Constant expressions which fold into constants are recognized as constant TRUE or FALSE. This feature facilitates keeping several versions of similar source in the same file without adding extra generated code after the code is compiled.

Example of Conditional Compilation

```
const
    version = 10;

if version = 7 then
    writeln('Too old!')
else
    writeln('Not too old!');
```

The code fragment above, with the constant "version" set equal to 10, has the same effect as a code fragment like this:

```
writeln('Not too old!');
```

Chapter 5 – Statements

Statements denote algorithmic actions, and are said to be executable. Statements define the actions that are to be performed on program objects that were introduced via type and variable declarations, discussed earlier in this manual.

5.1 Statement Labels

A statement can be *labelled* by preceding it with an unsigned integer constant in the range 0 .. 9999, followed by a colon. The statement can then be explicitly referred to by a **goto** statement.

5.1.1 Scope Of Statement Labels

The scope of a statement label is the body of the procedure or function in which the label is declared and all nested procedures and functions. This means that a **goto** statement cannot transfer control into a procedure or function body unless that procedure or function has been activated.

5.2 Assignment Statements

The *assignment statement* replaces the current value of a variable with a new value derived from expression evaluation, or defines the value that a function variable returns.

```
<assignment statement> ::=
                          <variable> := <expression>
                        | <function identifier> := <expression>
```

5.2.1 Assignments to Variables and Functions

The part to the left of the assignment symbol, :=, is evaluated to obtain a reference to some variable. The expression on the right side is evaluated to obtain a value. The referenced variable's current value is discarded and

replaced with the expression's value.

The variable on the left hand side of an assignment statement must be assignment compatible (see Chapter 2 – "Defining Data Types") with the type of the expression on the right hand side.

A string constant may be assigned to a variable of type **packed array [1..n] of char**, providing that the string value is the same length as the array object. The maximum length of such an assignment is 255 characters.

Examples of Assignment Statements

```
x := 5                      { simple assignment to variable }
y := x * 10 + 18            { assignment of expression      }
ch := CHR(10)               { assignment of function value  }
rope := 'hemp'              { string assignment             }
poke := POINTER($200)
poke^ := 0                  { clobber the system vector     }
```

5.3 Procedure Reference Statement

A procedure reference statement creates an environment for execution of the specified procedure and transfers control to that procedure.

```
<procedure call statement> ::=
                              <procedure identifier><actual parameter list>
                            |   <procedure identifier>

<actual parameter list>    ::=
                              (<actual parameter> {,<actual parameter>})

<actual parameter>         ::= <expression>
                            |   <procedure identifier>
                            |     <function identifier>
```

The actual parameter list must be compatible with the formal parameter list of the procedure. An actual parameter corresponds to the formal parameter which occupies the same ordinal position in the formal parameter list.

Only formal parameters that are value parameters can have an actual parameter which is an <expression>. Value parameters must be assignment compatible with the type of the formal parameter.

Formal parameters that are **var** parameters must have actual parameters that are identical types. In addition, the actual parameters must not be components of **packed** objects.

5.4 Structured Statements

Structured statements are constructs composed of statement lists. They provide scope control, selective execution, or repetitive execution of the constituent statement lists.

```
<structured statement> ::= <begin statement>
                         | <if statement>
                         | <while statement>
                         | <repeat statement>
                         | <for statement>
                         | <case statement>
```

5.4.1 BEGIN .. END — Compound Statements

A **begin** statement specifies execution of a statement list. Exit from the statement list is either through completing execution of the last statement in the statement list, or through explicit transfer of control.

```
<begin statement> ::= begin <statement list> end
<statement list>  ::= <statement> {; <statement>}
```

5.4.2 IF .. THEN .. ELSE Statements

The **if** statement specifies that another statement be executed (or not) depending on the truth (or falsity) of a conditional expression. If the value of the conditional expression is **true**, the statement is executed. If the value of the conditional expression is **false**, either no subsequent statement is executed, or the statement following an **else** clause is executed.

```
<if statement> ::=
        if <Boolean expression> then <statement>
    |   if <Boolean expression> then <statement> else <statement>
```

Because Pascal statements are open forms, it is possible to construct a chain of **else if** clauses to select "one out of many different conditions".

In common with similar languages, Pascal has what is called the "dangling else" problem. If an **if** statement contains another **if** statement as a subordinate, when an **else** clause is encountered, which **if** statement does the **else** clause apply to? In Pascal, the **else** clause matches the most recent **if** statement that does not have an **else** clause. One of the examples below clarifies this point.

Examples of If Statements

```
                        { example of a simple if statement }
if day in [Monday .. Friday] then
        Get_up_and_go
else
        Roll_over

                        { an if statement with a compound block }
if sun > yardarm then
        begin
                make_cocktails;
                prepare_snacks;
                relax
        end
else
        flog_on

                                        { an else if chain    }
if  weather = raining  then
        sleep_in
else if  lawn = wet  then
        clip_the_hedge
else if  grass > 6 then
        mow_the_lawn
else
        turn_on_lawn_sprinklers

                                        { A dangling else clause }
if condition_1 then          { 1 }
        if condition_2 then      { 2 }
                if condition_3 then  { 3 }
                        ..... statements .....
                else  { goes with statement 1 }
                        ..... statements .....
        else      { goes with statement 2 }
        ..... statements .....
else        { goes with statement 3 }
        ..... statements .....
```

5.4.3 CASE Statements

A **case** statement selects one of its component statements depending on the value of an expression. The expression is called the case *selector.* Each of the component statements is tagged with one or more simple scalar constants. The tags are called selection specifications (<selection specs> for short). If the value of the selector matches that of one of the statement tags, that

statement is executed. If the selector value matches none of the statement selection specifications, the statement (if any) following an **otherwise** symbol is executed.

Note that this Pascal implementation differs from the ISO standard in the provision of the **otherwise** clause. ISO Pascal has no provision for "what to do if none of the case selectors match the selector expression". Strict Pascal considers this situation a run-time error.

```
<case statement> ::= case <expression> of <cases>
                         {otherwise: <statement>} end

<cases>    ::= <a case> {<a case>}
<a case>   ::=
    <selection spec> {, <selection spec>} : <statement>;

<selection spec> ::= <scalar constant>
```

Case selectors and the statement tags must be non-**real** scalar types. In addition, the case selectors and the statement tags must be of assignment compatible types.

It must be stressed that the selection specifications which the component statements are tagged with are *not* labels in the Pascal sense, and as such, cannot be used as the target of a **goto** statement, and neither should they appear in any label declaration part.

Examples of Case Statements

```
case wine_type of

Champagne:
  Anything_goes;

Cabernet:
  Roast_Lamb;

Chardonnay:
  Veal_Piccata;

otherwise:
    Hamburger;
end;
```

5.4.4 WHILE .. DO Statements

A **while** statement controls repetitive execution of another statement until evaluation of a Boolean expression becomes false.

<while statement> ::= **while** <expression> **do** <statement>

The <statement> is repeated while the value of <expression> remains **true**. The <expression> must be of type **Boolean**. When <expression> becomes **false**, control passes to the statement after the **while** statement. If the value of <expression> is **false** at the time that the **while** statement is encountered for the first time, the subordinate statement is never executed at all. Thus the **while** statement provides a means to "do nothing gracefully". Contrast this behavior with the **repeat** statement described below.

Example of WHILE Statement

```
while bytes_to_go > 0 do
      begin
            if  bytes_to_go <= BlockSize then
                TransferLength := bytes_to_go
            else
                TransferLength := BlockSize;
            DoTransfer;
            bytes_to_go := bytes_to_go - TransferLength;
            BlockNumber := BlockNumber + 1
      end
```

5.4.5 REPEAT .. UNTIL Statements

The **repeat** statement controls the repetitive execution of a list of statements. The statements are executed until the condition at the end of the statement evaluates to **true**. The form of a **repeat** statement is:

<repeat statement> ::= **repeat** <statement list> **until** <expression>

The expression controlling repetition must be of type **Boolean**. The statement between the **repeat** and **until** symbols is executed repeatedly until the expression becomes **true**. Note that the body of a **repeat** statement is always executed at least once, since the termination test is at the end. Contrast this behavior with the **while** statement described in the previous subsection.

Example of Repeat Statement

```
repeat
      consume_glassfull;
      refill_glass;
until (Champagne_Volume <= 0) or (Consumer = Blotto);
```

5.4.6 FOR .. DO Statements

The **for** statement executes its subordinate statement repeatedly, while a progression of values is assigned to a *control variable* of the **for** statement.

```
<for statement> ::=
                  for <control variable> := <for list> do <statement>

    <for list>    ::= <initial value> to <final value>
                    | <initial value> downto <final value>

<control variable> ::= <identifier>

  <initial value>  ::= <expression>
    <final value>  ::= <expression>
```

The control variable is set to the initial value. After every iteration the control variable is either incremented (**to**) or decremented (**downto**) until its value is greater than or less than the final value.

The control variable, the initial value, and the final value, must all be of the same scalar type or a subrange of that scalar type. No part of the statement controlled by the **for** statement may alter the control variable during the execution of the **for** statement.

Neither the control variable, nor the initial value, nor the final value, may be of type **real**. The control variable must be local to the procedure or function that contains the **for** statement.

The value of the control variable is undefined on normal termination from the **for** statement. If the **for** statement is exited prematurely (via a **goto** statement), the value of the control variable is defined.

Examples of the FOR Statement

```
                        { initialize an array to zero }
for index := 1 to 100 do
    row[index] := 0

                        { scan from the end of an array }
for where := 200 downto 1 do
    if what[where] = thing then
        foundit := true
```

5.5 The WITH Statement

The **with** statement provides a "shorthand" notation for referring to fields in a record. The **with** statement effectively "opens the scope" that contains field identifiers of a specified **record** variable.

<with statement> ::=
with <record variable> {,<record variable>}
do <statement>

Within the body of the **with** statement, fields of the specified record variable do not need to be qualified by the name of the record.

If there is a local variable "x" and a field "x" in a record "r" which is the subject of a **with** statement, the statement:

with r **do**

"hides" the local variable "x" until the end of the **with** statement.

A **with** statement which has multiple <record variable> fields is interpreted as nested **with** statements. The statement:

with record_1, record_2, record_3 **do**

is equivalent to the statement:

```
with record_1 do
    with record_2 do
        with record_3 do
            ..... statement .....
```

Example of the WITH Statement

```
var
    TreeTop: SymTree;

    with TreeTop do
        begin
            LeftNode := nil;
            RightNode := nil
    end     { with }
```

This is a shorthand for the following statements

```
TreeTop.LeftNode  := nil;
TreeTop.RightNode := nil
```

5.6 The GOTO Statement

The **goto** statement names as its successor, a labelled statement designated by a label.

<goto statement> ::= **goto** <label>

The following should be noted concerning the **goto** statement and the label that it designates:

The scope of a label is the procedure in which that label is defined and all nested procedures and functions. Therefore it is not possible (nor valid) to jump into a procedure when no activation of the procedure exists.

Every label in a procedure must be declared in the label declaration part at the head of the procedure.

Example of Goto Statement

```
if  status = error  then
  goto 9999              { exit to end of procedure }
```

Chapter 6 – Input and Output

Input and Output facilities provide the means whereby a Pascal program can communicate with the world outside the computer system on which it runs.

SVS Pascal supports the input-output facilities as defined by standard Pascal, and additionally supports untyped (block access) files, interactive files, random access to typed files and *unit* input-output (direct access to the devices on the system).

6.1 General File Handling Procedures

This Section covers the standard Pascal procedures for handling files of any type. The four supplied procedures are GET, PUT, RESET and REWRITE.

6.1.1 The File Buffer Variable

A Pascal **file of** some_type is a *sequential* file – its components appear in strict sequential order (ignore the SEEK procedure for the duration of this discussion). Writing implies appending a component to the end of the file. Reading implies that the next component in sequence is obtained from the file. The following discussion applies only to typed files.

Associated with each typed file variable there is an implicit "buffer variable", often called the file "window". The buffer variable can be thought of as a place holder where the current file component is held. The buffer variable holds the next available component when reading. When writing, it holds the component that will be appended to the file by a PUT procedure call.

For a given file variable "f", the buffer variable is referenced by the notation "f^". Consider the following declarations:

```
type
    whammo = file of gobion;

var
    frammis: whammo;
    Curcomp: gobion;
```

When the file "frammis" is opened for reading via the RESET procedure call, the first component of the file is in the buffer variable. An assignment statement of the form:

```
CurComp := frammis^;
```

assigns the contents of the buffer variable to the variable "CurComp". The contents of the buffer variable then become undefined. The next component from the file is moved into the buffer variable by a GET procedure call.

When the file "frammis" is opened for writing via the REWRITE procedure call, the buffer variable is undefined. An assignment of the form:

```
frammis^ := CurComp;
```

assigns the value of the variable "CurComp" to the buffer variable. A subsequent PUT procedure call appends the contents of the buffer variable to the file "frammis". The contents of the buffer variable become undefined until another assignment defines it.

For files of type **interactive** the handling of the buffer variable is different. In standard Pascal, when a file is RESET, the first element of the file is read and placed in the file buffer variable. This means that the system would expect the user to type a character at the terminal, else the system would "hang". Thus a RESET on an **interactive** file does not perform an immediate GET. This affects the way that EOLN functions. When an end-of-line is read, EOLN becomes **true** and the character read is a space.

6.1.2 GET – Get Component from File

The procedure GET obtains the next element from a file (assuming there is a next element to be obtained). A call on the GET procedure of the form:

```
GET(file)
```

advances the current file position to the next component in the file. The value of this component is then assigned to the buffer variable file^.

If there was no "next component" in the file, the value of the buffer variable is undefined and the predicate EOF(file) becomes **true**.

If the predicate EOF(file) is already **true**, a GET(file) (in other words, trying to read past end-of-file) has an undefined result.

6.1.3 PUT – Append Component to a File

A call on the PUT procedure of the form:

PUT(file)

appends the value of the buffer variable file^ to the file "file". The value of file^ becomes undefined after the call to PUT. The predicate EOF(file) becomes **true** after the PUT.

If the predicate EOF(file) was **false** before the call to PUT (in other words, there were intervening GET's on the file), the call to PUT has an undefined result.

6.1.4 RESET – Open an Existing File

A call to the RESET procedure of the form:

RESET(file, string [, buffering option])

opens the file named 'string' and positions it at the beginning of the file. If the file variable had previously been opened, access to the previous file is lost, and no close or buffer flushing is done. For a 'proper' closing of any opened file, a specific call to CLOSE must be done. If the file is not empty, the first element of the file is assigned to the buffer variable file^ and the predicate EOF(file) becomes **false**. If the file is empty, the buffer variable file^ is undefined and the predicate EOF(file) becomes **true**.

If the file is an **interactive** file, RESET does not read the first element of the file.

SVS Pascal requires a second parameter to RESET. This parameter is the name of an **existing** disk file or device. The parameter takes the form of a **string** constant or variable.

The third parameter to RESET is an option to determine whether the file is buffered or unbuffered. The buffering option may be specified as the keyword BUFFERED or UNBUFFERED, and it is described in the subsection following REWRITE, below.

6.1.5 REWRITE – Create or Overwrite a File

The REWRITE procedure creates a new file of a specified name and discards any existing file of the same name. Thus a call of the form:

REWRITE(file, string [, buffering option])

discards the current value of the file variable "file", effectively creating a new file. The value of the buffer variable "file^" is undefined and the predicate EOF(file) becomes **true**.

If the variable had previously been opened, access to the previous files is lost, and no close or buffer flushing is done. For a 'proper' closing of any opened file, a specific call to close must be done.

SVS Pascal requires a second parameter to REWRITE. This parameter is the name of a disk file. The parameter can be a **string** variable or constant.

The third parameter to RESET is an option to determine whether the file is buffered or unbuffered. The buffering option may be specified as the keyword BUFFERED or UNBUFFERED, and it is described in the subsection below.

6.1.6 The Buffering Option on RESET and REWRITE

The optional "buffering option" parameter to RESET and REWRITE can be specified as either BUFFERED or UNBUFFERED. On some operating systems, there is a significant difference in throughput between buffered and unbuffered input output.

Normally, buffered input output is much more efficient than unbuffered input output. But, there can also be undesirable side effects in buffered input output, most notably that output does not appear at a terminal until a full buffer has been collected.

The "buffering option" parameter provides a means to request either buffered or unbuffered input output for the file specified in the RESET or REWRITE request. A given operating system might well override the request, depending on the nature of the device on which the file resides. The standard situation is unbuffered input output, in the absence of the "buffering option" parameter.

6.2 Text File Handling Procedures

Pascal provides standard procedures for controlling text-file input and output. These procedures apply to files of type **text** or **interactive**.

6.2.1 READ and READLN Intrinsics

READ and READLN read character strings representing numbers from a textfile and convert them into their internal representations. There is more on converting numbers later in this subsection.

READ (v_1, v_2, ..., v_n)

is equivalent to a

READ(input, v_1, v_2, ..., v_n)

READ (file, v_1, v_2, ..., v_n)

is equivalent to a sequence of READ procedure calls as follows:

READ (file, v_1);

READ(file, v_2); ....

READ(file, v_n)]

If "ch" is a variable of type **char**, the two programs displayed here are equivalent:

```
var                          var
    ch: char;                 ch: char;
  rasp: file of char;        rasp: file of char;

   READ(rasp, ch)             ch := rasp^;
                              GET(rasp)
 end                        end
```

If "v" is a variable of type **integer**, any subrange of **integer**, **real**, or **double**, the procedure reference:

READ(file, v)

reads a sequence of characters from the file referenced by "file". The sequence of characters should form a valid number according to Pascal's rules for numbers (described in Chapter 1). Note that if a **real** or **double** number contains a decimal point, there must be at least one digit on either side of the decimal point. When the number is formed it is then assigned to the variable "v". Blank lines and spaces preceding the number are skipped in the file. Reals are read in the same way as integers. Booleans cannot be read via a READ or READLN call. Structured types cannot be read.

If the sequence of characters read from the file do not form a valid number according to the syntax rules, one of two actions are taken: if I/O checking is on, the Pascal run-time system issues an error diagnostic; if I/O checking is off, READ or READLN return zero (0) and the IORESULT code is set. See Appendix A – "Messages from the Pascal System" for a list of I/O error codes.

6.2.2 READ from a file of any type

The READ procedure can also read from a file of any type. A READ procedure call of the form:

READ(file, v_1, v_2, ..., v_n);

is equivalent to the sequence:

```
v1 := file^; GET(file);
v2 := file^; GET(file);
   . . .
vn := file^; GET(file);
GET(file);
```

where the "v_n" are the list of variables to read into.

Note that the type of each variable in the list must be identical to the type of the elements in the file.

6.2.3 WRITE and WRITELN Intrinsics

The WRITE and WRITELN intrinsics append character strings to a textfile. Usually the character strings are generated by converting one or more *Write parameters* (see below) from their machine representations into external representations.

The procedure WRITELN differs from the procedure WRITE only in that WRITELN sends an end-of-line to the output file after the write is complete.

```
<write intrinsic>  ::= WRITE(<file> <write parameters>);
<writeln intrinsic> ::= WRITELN(<file> <write parameters>);
                      | WRITELN;
      <file>       ::= <file variable>,
<write parameters> ::= <write parameter> {, <write parameter>}
```

The <file> parameter in all cases is a file variable which refers to the file on which to append character strings. If the <file> parameter is omitted, output is written to file **output** (the computer standard output).

6.2.4 Write Parameters

The WRITE and WRITELN procedures can control the format of the individual elements that are written. Each parameter to WRITE or WRITELN

is of the form:

```
<write parameter>   ::= <element>
                      | <element>:<field width>
                      | <element>:<field width>:<fraction size>

    <element>       ::= is the value to be written.
                            (see descriptions below)

  <field width>     ::= <integer expression>

<fraction size>     ::= <integer expression>
```

<element> is the value to be written. It may be of type **char**, **integer**, **real**, **double**, **Boolean**, **string** or **packed array of char**.

<field width> and <fraction size> are optional. If <fraction size>) is present, <field width> must also be present.

<field width> specifies the size of the output field into which the converted value is written. If the converted value is smaller than <field width>, the field is filled out with leading spaces.

<fraction size> is only applicable when the <element> is of type **real** or **double** (see below).

6.2.4.1 Integer Element

The value of the **integer** expression is converted into a string representation of that expression in the base 10. The resulting string is placed right justified into the output field if a field width greater than needed is specified. If <field width> is too small to contain the resulting character string, the output field is expanded until it can contain the output string. If the integer expression is negative in value, a minus sign precedes the leftmost significant digit in the field. If the integer expression is positive, no space precedes the character string unless the <field width> is greater than the number of characters to be printed. If <field width> is omitted, the default field width is the minimum required to print the value.

6.2.4.2 Real or Double Element

A **real** or **double** element is converted much the same as an **integer** element, except that there can be a specification for the number of digits after the decimal point. In this case, <fraction size> specifies the number of digits to appear after the decimal point. The converted value is then written in so called "fixed point" notation. If <fraction size> is omitted, the converted number is written out in the floating or exponential notation. The diagram below illustrates the different forms of writing **real** elements.

```
WRITE(number:f)
        results in a number of the form:
 +x.yyyyyE +nn
```

where "f" is the total number of characters in the converted number. There is one digit before the decimal point and "f"-7 digits after the decimal point.

```
WRITE(number:f:w)
                results in a number of the form:
xxx.yyy
```

where "f" is the total number of characters (including the decimal point), and "w" is the number of digits after the decimal point.

The extreme **real** and **double** values are printed as follows: positive infinity prints as a row of + signs; negative infinity prints as a row of − signs; NaN (Not a Number) prints as a row of ? marks.

6.2.4.3 Scalar Subrange Element

A write parameter which is a scalar subrange is handled exactly as the scalar range of which it is a subrange.

6.2.4.4 Character Element

A write parameter which is a character is output as a single string character right justified in the output field. If <field width> is greater than one (1), the field is filled with leading spaces.

Furthermore, an <element> of type **char** means that the two programs displayed below are equivalent.

```
WRITE(file, <char expression>:<field width>)
                                        is equivalent to

file^ := ' '; {  these two statements repeated .....  }
PUT(file);    {  <field width> − 1 times               }

file^ :=   <char expression>;     PUT(file)
```

6.2.4.5 String Element or Packed Array of Char

A write parameter which is a **string** or **packed array of char** expression is placed right justified into the output field with leading spaces. If <field width> is less than the dynamic length of a **string** expression, the output field is expanded to contain the string. If <fieldwidth> is less than the length of a **packed array of char** expression, then only the first <fieldwidth> characters are output. If <field width> is omitted, the output field is the minimum

length needed to hold the string.

6.2.4.6 Boolean Element

An expression which is of type **Boolean** is written as one of the predefined identifiers **False** or **True**. If <field width> is greater than the length of the resulting string (5 for "**False**"; 4 for "**True**"), the string is written with leading spaces. If <field width> is less than the length of the string, the field is expanded to contain the string. If the value of the expression is not a valid **Boolean**, the string "UNDEF" is printed.

6.2.4.7 Hexadecimal Output

Integer expressions may also be output in hexadecimal representation. This is accomplished by appending the identifiers **hex** to the right of the value, or in the call that a field width is given, after the field width. Exactly <field width> characters are output. Since all such expressions are converted to type **longint** prior to output, a maximum of 8 hexadecimal digits are printed. Any extra characters are blank. If less than 8 characters are specified, then the least significant portion of the value is output. The default field width is 8.

6.2.4.8 Pointer Output

Pointer may also be written to **text** files. Their value is output in hexadecimal notation. An optional field width is accepted, the default width being 8.

6.2.5 WRITE to file of any type

The WRITE intrinsic can also write to a file of any type. A WRITE procedure call of the form:

WRITE(file, $expr_1$, $expr_2$, ..., $expr_n$);

is equivalent to the sequence:

```
file^ := expr1; PUT(file);
file^ := expr2; PUT(file);
    . . .
file^ := exprn; PUT(file);
```

where the $expr_n$ are a list of expressions to be written to the file.

Note that the type of each expression in the list must be the same as the type of the elements in the file. Integer subranges are converted to the proper

length as needed.

6.2.6 SEEK – Random Access to Typed Files

SVS Pascal supports random access to files of specific types. The SEEK procedure has two parameters, namely the file variable and an integer specifying the record number to which the file window should be moved. SEEK can only be applied to typed files that are not text files. The format of SEEK is:

procedure SEEK(file: file_type; position: **longint**);

file — is the file variable for the specified file.

position — is the number of the record to which the file window is to be moved. Records are numbered sequentially from zero (0).

SEEK moves the file window to the "position"th record in the file specified by "file". The EOF and EOLN predicates are set to **false**.

An attempt to PUT a record beyond the physical end of file sets the EOF predicate **true**. The physical end of file is the place where the next record in the file would overwrite another file on the storage device.

If a GET or PUT is not performed between two SEEK procedures, the contents of the file window are undefined.

6.2.7 CLOSE – Close a File

CLOSE removes the association of a file variable with an external file. A CLOSE procedure call marks the file as closed. The file variable for that file is then undefined. If a file is already closed, a CLOSE call does nothing. The form of the CLOSE procedure is:

procedure CLOSE(file [, close_option]);

file — is a file variable.

close_option — is an optional parameter that controls the disposition of the closed file. "close_option" can be one of the following:

normal — The state of the file is set to closed. If the file was opened with a RESET procedure call, the "normal" option means that the file is retained in the file system. If the file was opened with a REWRITE procedure call, the "normal" option means that the file is removed from the file system under operating systems where the old

file of the same name is still intact. The "normal" option is the default.

lock — makes the file permanent in the disk system if it is a disk file. Any existing file of the same name is removed from the file system. If the file is not a disk file, a "normal" close is done.

purge — deletes the file from the file system if the file is on a block-structured device. If the file associated with "file" is a device instead of a block-structured volume, the device is set off-line. If no physical device or file is associated with "file", a "normal" close is done.

crunch — is the same as the "lock" option but in addition, truncates the file at the point at which it was last accessed. That is, the end of the file is the position at which the last PUT or GET was performed. This option only works under certain operating systems.

6.2.8 PAGE — Skip to New Page

The procedure PAGE may be used to skip to the top of a new page on a **text** or **interactive** file. The form of PAGE is:

```
procedure page(file: text);
```

A call to PAGE actually does not guarantee that the device being written to will advance to a new page. Instead it outputs a single ASCII for feed character, 'OC', to the specified file. In most cases this will result in a form feed.

6.3 Block Input Output Intrinsics

BLOCKREAD and BLOCKWRITE support random (block level) access to untyped files only. A block is 512 bytes of data regardless of the actual file system blocking factor.

6.3.1 BLOCKREAD — Read Block from File

BLOCKREAD reads specific blocks from an untyped file. The function definition is:

```
function BLOCKREAD(file, where, blocks [,relblock]): integer;
```

file — is an untyped file.

where — is a variable of any type. The variable must be large enough to contain the number of blocks requested.

blocks — is an **integer** value which specifies the number of blocks to read from the file.

relblock — is an optional parameter. If "relblock" is present, it represents the block number at which to start reading from. Blocks are numbered relative to zero (0).

if "relblock" is omitted, it implies a sequential read of the next block in the file. When the file is opened, or when the file is reset, the starting block number is set to zero (0). Thus a BLOCKREAD with the "relblock" parameter omitted starts reading from block zero, and reads sequential blocks on every subsequent call that has the "relblock" parameter omitted.

The return value of BLOCKREAD is the number of blocks actually read. If the value is zero, it indicates either end-of-file or an error condition. If the value is greater than zero, it indicates the number of blocks read. If the return value is less than the number of blocks specified in the function call, it is possible that an end-of-file was encountered during the read.

6.3.2 BLOCKWRITE – Write Block to File

BLOCKWRITE writes specific blocks to an untyped file. The function definition is:

function BLOCKWRITE(file, where, blocks [,relblock]): **integer**;

file — is an untyped file.

where — is a variable of any type. It must be large enough to contain the number of blocks to be transferred.

blocks — is an **integer** value which specifies the number of blocks to write to the file.

relblock — is an optional parameter. If "relblock" is present, it represents the block number at which to start writing to. Blocks are numbered relative to zero (0).

if "relblock" is omitted, it implies a sequential write of the next block in the file. When the file is opened, or when the file is reset, the starting block number is set to zero (0). Thus a BLOCKWRITE with the "relblock" parameter omitted

starts writing to block zero, and writes blocks sequentially on every subsequent call that has the "relblock" parameter omitted.

The return value of BLOCKWRITE is the number of blocks that were actually written. If the return value is zero or a less than the number of blocks specified, it means either that there was an error or that there is no room for the blocks on the device.

6.4 IORESULT – Return Input-Output Result

IORESULT is a function that can be used after an input-output operation to check on the validity of the operation. The function definition is:

function IORESULT: **integer**;

Use of the IORESULT function is only appropriate if I/O checking has been turned off. The $I- compiler option turns checking off. If I/O checking is on (as it is by default) or turned on via the $I+ compiler option, any I/O error generates a non-recoverable run-time error.

If I/O checking has been turned off, I/O errors do not generate run-time errors, and the programmer can then use IORESULT to check the completion status of each input output operation.

The value of IORESULT is zero if an input-output operation has a normal completion. If the value is non-zero, it indicates some form of error has occurred. See Appendix A – "Messages from the Pascal System" for a list of error codes.

Example of using IORESULT

```
{$I-}  { Turn off the I/O Checking }
type
   data_file = text;
var
   data: data_file;

   RESET(data, '/source/printfile');
   if IORESULT <> 0 then begin    { <> 0 = problem }
      REWRITE (data, '/source/printfile'); { so create it }
      if IORESULT <> 0 then begin
         WRITELN('Cannot create /source/printfile');
         HALT
      end;
   end;
```

In the above example, the $I- comment toggle turns off the I/O checking for that part of the program. The IORESULT function returns a non-zero value to

mean that the file could not be RESET, so the program then tries a REWRITE statement. If that fails, then the program halts.

Chapter 7 – Program Structure

A Pascal program is a collection of declarations and statements which is meant to be translated, via a compilation process, into a relocatable object-module. Object modules obtained from other, separate compilations can be combined, via a linking process, into a form suitable for execution.

The collection of declarations and statements may also include compiler directives which control the compilation, and do not change the meaning of the program.

The results of compilations, the object modules, are sometimes referred to as ".obj" files since this is the normal file name extension for such files. SVS Pascal is very flexible in the mechanisms for creating ".obj" files which are not complete executable programs and combining them in the linking process. The **unit** mechanism, derived from UCSD Pascal, is provided for "secure" independent compilation. Using this mechanism, a group of declarations and procedures can be compiled into an ".obj" file. This ".obj" file can be used by other Pascal compilations to insure that interfaces are consistent, and subsequently linked with the ".obj" files created in these compilations. Alternatively, independent compilation via the **external** (or **cexternal**) mechanism can be used for "insecure" independent compilation of Pascal routines, or for linking Pascal to routines written in other SVS languages or assembly language.

7.1 Compilation Units

Before describing in detail the various compilation units and their components, the following are some examples of compilation units with accompanying explanations.

Example of Complete Program Compilation Unit

```
program complete;
   var i: integer;
begin
   i := 17;
   writeln(i);
end.
```

The above program is complete and can be compiled and executed. It does not make use of any separate compilation.

Example of Program with Insecure Separate Compilation

```
program missingsomething;
   var i: longint;

   procedure getvalue(var fi: longint); external;

   procedure callme;
   begin
      writeln('I got called!');
   end;

begin
   getvalue(i);
   writeln(i);
end.
```

This example illustrates the call on an external procedure called getvalue which will have to be supplied in the linking process in order to make a complete executable program. It is possible that this external procedure has been written in assembly language, or in SVS FORTRAN, or in Pascal. (Note: if the procedure had been written in SVS C, the calling sequence to the external would have to be different and the programmer should have coded getvalue as a **cexternal** instead of an **external**).

Regardless of the origin of getvalue, the Pascal system will make no attempt to match parameter types, etc. between the call and the code called. Pascal is satisfied that the **external** declaration describes the interface and it is the programmer's responsibility to insure that the receiving subroutine is suitable. Thus, this method of independent compilation is referred to as "insecure".

The example also contains a procedure "callme" which may well be referenced in some other compilation unit as an **external**. This other compilation unit must not, however, contain a main program, since it is not allowed to link together ".obj" files containing more than one main.

Example of a Simple Unit

```
unit IHideAndHoldAndPrintX;
interface
   var publicinteger: integer;

   procedure setx(fx: real);
   procedure printx;

implementation
   var x: real;

   procedure setx;
   begin
      x := fx;
   end;

   procedure printx;
   begin
   writeln(x);
   end;
end.
```

This example creates a **unit** with two procedures in its public part and a private variable. The ".obj" created by the Pascal system for this unit contains linkable object code for the two procedures and contains the *source code* for the interface section of the unit. Let us assume that the created object code for this unit is named "hide.obj".

When another compilation "uses" this unit (see example below), which is to say uses the ".obj" code of this unit, the interface source code declarations are extracted from the unit's ".obj" file and processed to insure that interfaces match properly. This is why the **unit** mechanism is referred to as "secure" independent compilation.

Example of Program Using a Unit

```
program UseUnit;
   uses {$U hide.obj} IHideAndHoldAndPrintX;
begin
   publicinteger := 99;
   setx(17.3); printx;
   writeln(publicinteger);
end.
```

This program has available the variables in the interface section of the referenced unit as well as the procedures declared there (by actual inclusion of the source code which is part of the unit's ".obj" file). The Pascal system

checks and enforces that the interfaces are matching between this compilation and the unit.

The Pascal system must be told what the file name of the ".obj" of the referenced unit is. This is done using the $U directive. In the event that more than one unit is to be used, the following method should be utilized:

```
uses  {$U file1.obj}  FirstUnit,
      {$U file2.obj}  SecondUnit;
```

The order in which these units are used may be important. If the SecondUnit unit used FirstUnit when it was compiled, it more than likely depended on FirstUnit to make its own declarations meaningful. In this event, the order must be as shown.

The examples shown here illustrate only a few of the possibilities. Units can use units. Global procedure and function names in programs and interface procedures and functions in units become available for reference via the external mechanism, etc.

The key to properly using units is to remember that the interface information is included in the using compile as source declarations. This fact determines the order in which compilation must be done and what must used where and in what order.

The more formal details of compilation units follows.

A compilation unit is either a **program** (a main program), or a **unit**. A complete executable program consists of a single **program** and zero or more **units**.

A **program** is a main program, consisting of all the statements between a **program** statement and an **end.** statement. The main program is described in more detail later in this chapter, in the section entitled "Program Heading".

A **unit** is a collection of declarations and statements packaged so as to make parts of the declarations in the unit public to other parts of the same compilation unit or separate compilation units. Units are useful for sharing common code among different programs or as a means to avoid compiling a huge program every time one line is changed. Units are compiled separately.

A program or unit that uses another unit is known as a "host". A host uses other units' declarations by naming those units in **uses** declarations. The **uses** clause appears after a program heading or it appears in a **unit** at the start of the **interface** section (see below).

A unit contains two major parts, namely an **interface** part which describes how other units view this unit, and an **implementation** part which supplies the

actual body of code to implement this unit.

```
<unit>          ::= unit <identifier>;
                    <interface part>
                     <implementation part>
                    end.

<interface part>    ::= interface
                     <uses clause>
                      <constant definition part>
                       <type definition part>
                        <variable definition part>
                         <procedure and function declaration part>

<implementation part> ::=
                    implementation
                     <label declaration part>
                      <constant definition part>
                       <type definition part>
                        <variable definition part>
                         <procedure and function declaration part>

<uses clause>       ::= uses <identifier> {,<identifier>} ;
```

The **interface** part declares constants, types, variables, procedures and functions that are globally available. A host program that uses that unit has access to those objects just as if they had been declared in the host program itself.

Procedures and functions declared in the **interface** part consist only of the procedure or function name and the description of the formal parameters. These declarations serve as procedure or function *prototypes* – there is no executable code associated with them. This is equivalent to a **forward** declaration except that no **forward** attribute is allowed.

The **implementation** part follows the **interface** part. Local objects are declared first, then the global procedures and functions are declared. Formal parameters and function result type declarations are omitted from the **implementation** part, since they were already declared in the **interface** part.

A unit can consist entirely of interface declarations (constants, types and variables). There need not be any procedure or function declarations.

The declarations in the **interface** part of a **unit** are accessible in another compilation only after that **unit** is specified in a **uses** statement of that compilation. The **uses** clause is used in conjunction with the $U compiler option. The unit will be searched for the file specified in the most recently appearing $U option. The file searched will be the file name with and ".obj" suffix. Thus the unit must have been previously compiled.

The overall layout of a **unit** is like this:

```
unit   GanipGanop;

interface          { This part declares the    }
                   { interface section         }

uses  names of     { This part is optional if                }
      other units  { GanipGanop does not use any             }
                   { things from other units                 }
                   { Note that if any declarations           }
                   { imported from other units are           }
                   { referenced in the interface             }
                   { part of GanipGanop then the             }
                   { compilation that uses                   }
                   { GanipGanop must first uses              }
                   { that other unit.
{..... declarations and
        procedure headings
          for the GanipGanop unit.
            All these declarations and procedure
              headings are PUBLIC to other units .....}

implementation     { This part declares the    }
                   { implementation section    }
{..... declarations and
        code for the GanipGanop unit.
          All these declarations and code are
            PRIVATE to GanipGanop .....}
end.    { of the GanipGanop unit }
```

7.2 Declarations and Scope of Identifiers

Declarations introduce program objects, together with their identifiers, which denote these objects elsewhere in a program.

```
<declaration> ::= <label declaration>
                | <constant declaration>
                | <type declaration>
                | <variable declaration>
                | <procedure or function declaration>
```

The program region (over which all uses of an identifier are associated with the same object) is called the *scope* of the identifier. Within a compilation unit, such a region is either a unit body or a block body. In the case of a **unit**, the scope is a declaration list. In the case of a block, the scope is a statement list preceded by an optional declaration list.

The scope of an identifier is determined by the context in which it was declared.

A program or a unit is a static construct intended to control the scope of identifiers according to these rules:

1. The scope of an identifier declared at the outermost level of a program is the body of that program.
2. The scope of an identifier listed in the **interface** part of a unit is the body of that unit, and is also extended "outwards" to any other unit that uses that unit.
3. Identifiers declared at the outermost level of the **implementation** part of a unit have the entire body of that unit as their scope, but are private to that unit.

Procedure or function blocks also control the scope of identifiers. There are both similarities with, and differences from, programs or units.

Like programs or units, blocks control the scope of identifiers.

Unlike programs or units, blocks control the processing of declarations and determine when the declarations take effect.

The block-structured scope rules are as follows:

1. The scope of an identifier declared in the declaration list of a block is the body of that block.
2. If the scope of an identifier includes another block, its scope is extended "inward" to include the body of that inner block, unless the body contains a re-declaration of that identifier.
3. An identifier which is declared as a formal parameter of a procedure or function has as its scope the body of that procedure or function.
4. Field selectors are identifiers introduced as part of the definition of a record type for the purpose of selecting fields of records. The scope of a field selector is the record in which it is declared. As with the nesting of procedures, the existence of an inner scope identifier masks the accessibility of any outer identifiers of the same name. Field selectors must be unique within the declaration of a record.
5. Identifiers must be unique within the bounds of a given scope.

7.3 Program Heading

The **program** statement identifies the main program for a Pascal compilation. In SVS Pascal, the program header is scanned but otherwise ignored. A program has the same form as a procedure declaration except for the heading.

<program> ::= <program heading> {<uses clause>} <block>.

<program heading> ::=
program <identifier> {(<program parameters>)};

<program parameters> ::= <identifier> {,<identifier>}

The identifier following the word **program** is the program name. It has no further meaning inside the program. The program parameters are optional. No global identifiers in the program may have the same name as any of the program parameters.

7.3.1 Predeclared Variables

SVS Pascal supplies five pre-declared variables. First there are standard files :

input is the standard file from which console input can be done via READ and READLN statements,

output is the standard file to which console output is directed via WRITE and WRITELN statements,

stderr is the standard error output file. On those operating systems which support a separate file for error responses, **stderr** is connected to that stream. On those operating systems which do not support a separate file for error responses, **stderr** is connected to the same place as **output**.

Then there are the two variables associated with obtaining arguments from the operating system command line (see the next subject heading below):

argc is a count of the number of arguments supplied on the command line.

argv is an array of pointers to the character strings containing the command line arguments.

7.3.1.1 ARGC and ARGV — Access to Command Line

As mentioned above, "argc" and "argv" provide access to the Pascal program's command line as the user typed it. "argc" and "argv" can be considered to be defined by a declaration of the form:

```
type
   stringtype = string[anylength];
   pstring = ^stringtype;

var
   argc: integer;
   argv: array[1 .. argc] of pstring;
```

Each element of argv contains a separate field from the command line that invoked this Pascal program. If "argc" is zero (0), no attempt should be made to reference "argv". The first element of "argv" is the first parameter from the command line. The name of the command itself may or may not be available as the first command line argument depending on the operating system under which the program is run. Avoid assigning to any element of "argv".

7.4 Declarations

7.4.1 Label Declarations

The label declaration part declares all labels (which tag statements) in the statement part of the block.

<label declaration part> ::= **label** <label> {, <label>};

7.4.2 Constant Definition

The constant definition part declares all constant names and their associated values that are local to the procedure or function definition.

<constant definition part> ::= **const** <constant definition list>

<constant definition list> ::=
<constant definition> {<constant definition>}

7.4.3 Type Definition

The type definition part contains all the type definitions that are local to the procedure or function definition.

<type definition part> ::= **type** <type definition list>

<type definition list> ::= <type definition> {<type definition>}

7.4.4 Variable Declaration

The variable declaration part contains a definition of all the variables that are local to the procedure or function.

<variable declaration part> ::= **var** <variable declaration list>

<variable declaration list> ::=
<variable declaration> {<variable declaration>}

7.5 Procedure and Function Declaration

A *procedure declaration* or a *function declaration* associates an identifier (the procedure or function name) with a collection of declarations and statements. A Pascal statement can then cause that procedure to be executed (activated) by giving its name in a *procedure reference* statement. A function declaration is similar to that of a procedure with the additional capability that a function can compute and return a value, called the value of the function. A function is referenced by giving its name in an expression, when the value of the function appears as a factor in that expression.

The type of value that a function returns is specified when the function is declared. The function return value is the value last assigned to its function identifier before a return is made from the function. Returning from a function without ever assigning a value to the function designator (for the current activation of the function) produces an undefined result (usually with undesirable results from the programmer's viewpoint).

Using a procedure or function identifier within the declaration of that procedure or function implies recursive activation of that procedure or function, except when a function identifier appears on the left hand side of an assignment statement, (implying assignment to the function variable rather than recursive activation – see below).

<procedure declaration> ::= <procedure heading> <block>

<block> ::= <label definition part>
<constant definition part>
<type definition part>
<variable declaration part>
<procedure and function declaration part>
<statement part>

<statement part> ::= **begin** <statement list> **end**

<statement list> ::= <statement> {; <statement>}

All the definition and declaration parts above are optional, with the exception of the <statement part>.

The *procedure heading* specifies the identifier that names the procedure, and any formal parameters for that procedure.

Procedure parameters are either *value* parameters, *variable* parameters, or *procedure* or *function* parameters.

```
<procedure heading> ::=
  procedure <identifier>; {<attribute>;}
| procedure <identifier> (<formal parameters>); {<attribute>;}

<function heading> ::=
    function <identifier>: <result type>; {<attribute>;}
|   function <identifier>(<formal parameters<);{<attribute>;}

<formal parameters> ::=
          <formal parameter> {;<formal parameter>}

<formal parameter> ::=
                    <parameter group>
                  | var <parameter group>
                  |   <procedure heading>
                  |     <function heading>

<parameter group> ::=
                  <identifier> {,<identifier>}:<type identifier>

    <attribute>   ::= external | forward | cexternal

  <result type>   ::= <simple type>
```

Note that the "external", "forward", and "cexternal" attributes are optional.

7.5.1 External and Forward Attributes

A Pascal host can use routines that are separately compiled or assembled in languages other than Pascal. To use an external routine, the host must make a **procedure** or **function** declaration for that external routine just as if it is a Pascal routine that is declared in this compilation unit or another compilation unit. The declaration is then followed by the **external** attribute to indicate that the body routine does not appear in the current compilation unit. External routines must conform with the Pascal calling conventions and data representation methods as defined in Appendix E – "Data Representations". The **cexternal** attribute means that the compiler generates calls to external procedures in a manner which is compatible with the SVS C compiler.

Pascal normally dictates that procedures and functions be declared before they can be referenced. There are cases when program layout makes this impossible, such that a procedure or function must be referenced before it can be declared. The **forward** attribute indicates that the particular procedure or function declaration consists only of the header, and that the body of that

function declaration consists only of the header, and that the body of that procedure or function appears later in the program source text, possible after it is referenced. A forward-declared procedure or function, then, is actually declared in two distinct parts: its header or prototype is declared, with the **forward** attribute, before any reference is ever made to it; at some later point in the program source text, its body is declared. At this later point, the formal parameter section must not appear.

7.5.2 Parameters for Procedures and Functions

Parameters (also called arguments) provide a dynamic substitution method such that a procedure or function can process different sets of data in different activations.

There is a correspondence between the *formal* parameters declared in a procedure or function heading and the *actual* parameters supplied when the procedure or function is activated.

The procedure or function heading declares a list of *formal parameters.* These are "dummy" variables that are assigned values when the procedure or function is activated.

A reference to the procedure or function supplies a list of *actual parameters* that are substituted for the formal parameters, which then become local variables initialized to the value of the actual parameters.

There are four kinds of formal parameters:

- Value parameters.
- Variable or Reference parameters.
- Procedure parameters.
- Function parameters.

A parameter group without a preceding specifier, implies that the parameter is a value parameter.

7.5.2.1 Value Parameters

Value parameters are those whose formal parameter declaration has no symbol marking them as one of the other three forms. The corresponding actual parameter must be an expression. In the body of the procedure or function, the formal parameter is initialized to the value of the expression at the time the procedure or function is activated. The formal parameter is then just like a local variable. The value of the formal parameter may be changed by assignment – the actual parameter remains unchanged.

7.5.2.2 Variable Parameters

Variable parameters, also called *reference parameters*, are those whose declarations start with the symbol **var** (for variable). The actual parameter must be a variable of a type which is identical to that of the formal parameter. The formal parameter directly represents, and can change, the actual parameter's value during the entire execution of the procedure or function.

var actual parameters must be distinct actual variables. It is a programming error to supply the same variable to more than one actual parameter in a procedure or function reference.

All index computations, field selection and pointer dereferencing are done at the time the procedure or function reference is made.

7.5.2.3 Procedure and Function Parameters

Procedure and Function parameters are the names and parameter lists of procedures or functions that can be referenced by the current procedure.

These parameters are indicated by the symbol **procedure** or **function** in the formal parameter declarations. Such procedures or functions are called *parametric*. Actual parameters to parametric procedures and functions must be of identical type to those declared in the formal parameter declarations.

Examples of Procedure and Function Declarations

```
                        { a procedure with only value parameters }
procedure ByTheBook(Chapter, Verse: integer);
begin
    Chapter := 1;       { does not change the caller's
                            version of Chapter }
end;

                        { a procedure with variable parameters }
procedure Change(var winds: integer);
begin
    winds:= 76;         { Changes the caller's version }
end;
```

```
                        { the Ackerman function }
function  Ackerman(m, n: integer):integer;
begin

    if  m = 0  then
        Ackerman := n + 1
    else if  n = 0  then
        Ackerman := Ackerman(m - 1, 1)
    else
        Ackerman := Ackerman(m - 1, Ackerman(m, n - 1))
end;

                        { parametric function parameter }
function  Integrate(lo, hi: real;
                    what(x: real):real): real;
var
    start: integer;
    finish: integer;
    point: integer;
    current: real;
    sum: real;

begin
    start := TRUNC(lo);
    finish := ROUND(hi);
    sum := 0.0;

    for  point := start to finish do
    begin
        current := point;
        sum := sum + what(current);
    end;
    Integrate := sum / (finish - start);
end;
```

Chapter 8 – Standard Procedures and Functions

SVS Pascal (in common with other Pascal implementations) supplies a number of standard ("built in") procedures and functions. This chapter covers those. The standard procedures and functions fall into several logically related groups, as follows:

- String Manipulation. These intrinsics handle the SVS Pascal dynamic string types.
- Memory Management. These intrinsics deal with dynamic memory allocation and de-allocation.
- Arithmetic Functions.
- Boolean Predicates.
- Conversion Functions.
- Miscellaneous Low Level Procedures and Functions.

8.1 String Manipulation Facilities

This section discusses those facilities for manipulating **string** data types in Pascal. For purposes of this section, string datatypes are those declared **string[n]**, for some n, not **packed array[1..n] of char**. The type "stringtype" utilized below should be read as matching any type declared **string[n]**. Here is a brief summary of the facilities:

CONCAT	concatenate a number of strings into one string.
COPY	extract substring of a string.
DELETE	delete characters from a string.
INSERT	insert characters into a string.
LENGTH	determine the current dynamic length of a string.

POS	scan for a pattern within a string.
SCANEQ and SCANNE	scan for a specific character within a string.

8.1.1 LENGTH — Determine String Length

LENGTH is an **integer function** that returns the length of a string expression. The function definition is:

```
function LENGTH (source: stringtype): integer;
```

LENGTH returns an integer value which is the dynamic length of the string "source".

The length of the string '' is zero (0).

Examples of LENGTH

```
alphabet := 'abcdefghijklmnopqrstuvwxyz';
WRITELN(LENGTH(alphabet), '    ',
        alphabet[1], '    ',
        alphabet[LENGTH(alphabet)], '    ',
        LENGTH(''));
```

the following output is displayed

```
26          a       z       0
```

8.1.2 COPY — Copy a Substring

COPY returns a stringtype which is a substring of another string. The function definition is:

```
function COPY(source: stringtype;
              index: integer;
              size: integer): stringtype;
```

COPY returns a string which is a substring of the string "source". COPY extracts "size" characters from "source", starting at the character position given by "index".

The first character in the string is numbered 1.

If "index" is negative or zero, the result is a null string.

If "index" is greater than LENGTH(source), the result is a null string.

If "index" + "size" is greater than LENGTH(source), the result is a string which extends from "index" to LENGTH(source).

Example of COPY

```
var
   left: string[100];
   middle: string[100];
   right: string[100];
   title: string[255];

   title := 'Left Side.  Middle Part.  Right Side.';
   left := COPY(title, 1, 10);
   middle := COPY(title, 12, 12);
   right := COPY(title, 25, 11);
   WRITELN(left);
   WRITELN(middle);
   WRITELN(right);
```

This should generate the output:

```
Left Side.
Middle Part.
Right Side.
```

8.1.3 CONCAT – Concatenate Strings

CONCAT returns a stringtype result, which is the concatenation of its (string) parameters. The function definition of CONCAT is:

```
function  CONCAT (s1: stringtype;
                  s2: stringtype; .....
                  sn: stringtype): stringtype;
```

Each of the "Sn" is a string variable or a string constant or a literal value. There may be any number of source strings, each separated by a comma from the next. There must be at least two source strings.

Example of CONCAT

```
title := CONCAT('Here', ', there', ', and everywhere');
WRITELN(title);
```

This should generate the output:

```
Here, there, and everywhere
```

8.1.4 POS – Match a Substring in a String

POS is used for string matching. The function definition is:

```
function  POS (pattern: stringtype;
               inwhat: stringtype): integer;
```

POS scans from left to right trying to find an instance of the string "pattern" in the string "inwhat". If a match is found, POS returns an integer value that is the position in "inwhat" at which the "pattern" starts to match.

If there is no match, the result is zero (0).

If "pattern" is longer than "inwhat", the result is zero (0), or no match.

If no "pattern" is the null string, '', the result is one (1), since the null string matches the first position in any string.

Example of POS

```
herbs := 'Basil, Chervil, Fennel, Tarragon';
WRITELN(POS('Chervil', herbs), '  ', POS('Nutmeg', herbs));
```

This should generate the output:

```
8          0
```

8.1.5 SCANEQ and SCANNE – Scan for Character

SCANEQ and SCANNE search a character array until they find (SCANEQ) or do not find (SCANNE) a specified character in the array. The function definitions are:

```
function  SCANEQ(len: integer;  what: char; object): integer;
function  SCANNE(len: integer;  what: char; object): integer;
```

SCANxx scans "object" for "len" characters, or until the character "what" is found (SCANEQ) or not found (SCANNE). The result is the offset into "object" where the scan stopped. If the character "what" is not found (SCANEQ) or is found (SCANNE), SCANxx returns the value "len". If the "len" parameter is positive, scanning is from left to right; if the "len" parameter is negative, the scan proceeds from right to left, and a negative value is returned.

Note that the SCANxx functions simply look at bytes in memory. They ignore any higher level structure that the user might perceive or might have imposed on the object. Thus "object" is simply an address in memory at which to begin scanning (or in the case where "len" is negative, to end the scan). Thus, for example, if the programmer were to do a SCANEQ on a data type of **string**[80], the length byte of that string would also be scanned, and the results

might be unexpected.

8.1.6 DELETE – Delete Characters from String

DELETE removes a specified number of characters from a string. The procedure definition is:

```
procedure DELETE (destination: stringtype;
                  index: integer;
                  size: integer);
```

"destination" is a string. "index" and "size" are integers.

DELETE removes "size" characters from "destination", starting at the position specified by "index".

If "index" is greater than LENGTH(destination), there is no action taken.

If either "index" or "size" is negative or zero, there is no action taken.

If "index" + "size" is greater than LENGTH(destination), DELETE removes all characters from "index" up to the end of the "destination" string.

Example of DELETE

```
var
  large: string[100];

  large := 'A long exhausting rally, eh what, chaps';
  DELETE(large, 8, 11);
  WRITELN(large);
```

This should generate the output:

```
A long rally, eh what, chaps
```

8.1.7 INSERT – Insert Characters into String

INSERT inserts one character string into another character string at a specified place. The procedure definition is:

```
procedure INSERT (source: stringtype;
                  destination: stringtype;
                  index: integer);
```

The "source" string is inserted into the "destination" string at a position determined by the value of "index".

If the length of "destination" is less than the value of "index", then no action is taken.

No check is made as to if the length of the result string is greater than the maximum length of the destination string. The result in such a case is usually not what the programmer intended.

8.2 Storage Allocation Procedures

Dynamically allocated storage is held in a large common storage pool, called a "heap". Storage is allocated from that pool by using the procedure NEW. Storage is released back to the pool (de-allocated) by using the DISPOSE procedure. Alternatively, some Pascal implementations handle memory de-allocation via the MARK and RELEASE procedures. SVS Pascal provides MARK and RELEASE for compatibility.

NEW	is responsible for allocating storage.
DISPOSE	is responsible for freeing or releasing storage back to the common storage pool.
MARK	provides a means to "remember" the current top of the heap.
RELEASE	releases memory from a previously MARK'ed point.
MEMAVAIL	determines the amount of memory available for allocation.

8.2.1 NEW — Allocate Storage

The procedure NEW allocates dynamically available storage. If "p" is a variable of type pointer to "T", NEW(p) allocates storage for a variable of type "T" and assigns a pointer to that storage to the variable "p". There are two forms of the NEW procedure reference:

NEW(p)	allocates a new variable "v", and assigns the pointer reference of "v" to the pointer variable "p". If the type of "v" is a variant record, storage is allocated for the largest variant of the record. Storage for a specific variant can be allocated by using the second form of the NEW procedure, as follows:
NEW(p, t_1, t_2,..., t_n)	allocates a variable of the variant, with tag fields t_1 .. t_n. The tag fields must be listed contiguously and in the order of their declaration in the variant record type definition.

If NEW is used to allocate storage for a specific variant record, the subsequent call to DISPOSE must use exactly the same variant. Any mismatch between the variants specified on the call to NEW and those on the DISPOSE call can damage the integrity of the heap, causing strange behavior at best and system crashes at worst.

If NEW fails to allocate the requested storage (usually because the storage is not available), the pointer variable "p" contains the value **nil** upon return from the procedure.

Example of NEW

```
const
   UpperLimit = 255;

type
   LArray = array[1 .. UpperLimit] of integer;
   ArrayAddr = ^LArray;

var
   head: ArrayAddr;

   NEW(head);
   if head = nil then
      ..... take some recovery action .....
   else
      begin
         head^[1] := 0;    { zero fill array }
         MOVELEFT(head^[1], head^[2],
            SIZEOF(integer)*(UpperLimit - 1));
               .....    and so on .....
      end
```

8.2.2 DISPOSE — Dispose of Allocated Storage

DISPOSE frees (or de-allocates) dynamically allocated storage. The procedure reference:

```
DISPOSE (p);
```

frees up the allocated storage referenced by the pointer variable "p". Upon return from DISPOSE, the pointer variable "p" contains the value **nil**.

Attempts to DISPOSE using a pointer variable that contains **nil** is a no-op and is ignored.

If NEW was used to allocate a variable with a specific variant, DISPOSE should be called with exactly the same variant, else the heap is likely to be corrupted.

DISPOSE currently does not return the deallocated memory to the heap.

8.2.3 MARK — Mark Position of Heap

MARK is used to "remember" the current position of the top of heap. MARK and RELEASE are used together to de-allocate memory and return the

top of the heap to a previously MARK'ed point. For example, a procedure might, upon entry, MARK the heap top, then allocate large numbers of variables, and then, just prior to exiting, RELEASE all the allocated memory. Such a situation might occur, for instance, in allocating the local symbol table for an assembly unit. At the end of the unit, all the local labels need to disappear — MARK and RELEASE provide a handy means to dispose of storage in bulk. The procedure definition of MARK is:

```
procedure MARK(HeapPointer: ^anything);
```

"HeapPointer" must be a pointer — the pointer type is irrelevant but conventionally it is a pointer to a **longint**. "HeapPointer" must not be used for any purpose other than as a MARK pointer.

8.2.4 RELEASE — Release Allocated Memory

RELEASE is used to cut the heap back to a point previously MARK'ed. The procedure definition of RELEASE is:

```
procedure RELEASE(HeapPointer: ^anything);
```

As for MARK, "HeapPointer" is a pointer of any type but conventionally is a pointer to **longint**. RELEASE cuts the heap back to the place indicated by "HeapPointer". "HeapPointer" must have been properly initialized by a previous call to MARK. MARK's and RELEASE's must be matched properly.

8.2.5 MEMAVAIL — Determine Available Memory

MEMAVAIL returns the number of bytes available for allocation in the storage pool. The function definition of MEMAVAIL is:

```
function MEMAVAIL: longint;
```

8.3 Arithmetic Functions

8.3.1 ABS — Compute Absolute Value

ABS(x) computes the absolute value of its argument "x". The type of the result is the same as the type of "x", which must be either **integer**, **real**, or **double**.

8.3.2 SQR — Compute Square of a Number

SQR(x) computes the square of "x", that is, it computes x*x. The type of the result is the same as the type of "x", which must be either **integer**, **real**, or **double**.

8.3.3 SIN – Trigonometric Sine

SIN(x) computes the trigonometric sine of the argument "x". The type of "x" may be either **integer**, **real**, or **double**. The return type of SIN is always **real** or **double**. The argument is in radians.

8.3.4 COS – Trigonometric Cosine

COS(x) computes the trigonometric cosine of the argument "x". The type of "x" may be either **integer**, **real**, or **double**. The return type of COS is always **real** or **double**. The argument is in radians.

8.3.5 ARCTAN – Trigonometric Arctangent

ARCTAN(x) computes the trigonometric arctangent of the argument "x". The type of "x" may be either **integer**, **real**, or **double**. The return type of ARCTAN is always **real** or **double**.

8.3.6 EXP – Compute Exponential of Value

EXP(x) computes the exponential of the argument "x". The type of "x" may be either **integer**, **real**, or **double**. The return type of EXP is always **real** or **double**.

8.3.7 PWROFTEN – Compute Ten to a Power

The function PWROFTEN(x) returns a value which is 10 raised to the power specified by the argument. The function definition is:

```
function pwroften (exponent: integer): real;
```

The valid range of the "exponent" argument is 0 .. +38.

8.3.8 LN – Natural Logarithm of Value

LN(x) computes the natural logarithm of the argument "x". The type of "x" may be either **integer**, **real**, or **double**. The return type of LN is always **real** or **double**. It is an error to supply an argument less than or equal to zero.

8.3.9 SQRT – Square Root of Value

SQRT(x) computes the square root of the argument "x". The type of "x" may be either **integer**, **real**, or **double**. The return type of SQRT is always **real** or **double**. It is an error to supply an argument less than or equal to zero.

8.4 Predicates or Boolean Attributes

8.4.1 ODD – Test Integer for Odd or Even

ODD(x) determines if the argument is odd or even. The type of the argument "x" must be **integer**. The result is **true** if "x" is an odd number, **false** if "x" is an even number.

8.4.2 EOLN – Determine if End of Line Read

The function EOLN returns **true** if the textfile position is at an end-of-line character. Otherwise the EOLN function returns **false**. EOLN is only defined for files whose components are of type **text** or **interactive**.

8.4.3 EOF – Determine if End of File Read

The function EOF returns **true** if a read from a file encounters an end-of-file. EOF returns **false** in all other cases. To set EOF **true** for a file attached to the console, the EOF character must be typed. In SVS Pascal this is Control-D. For a textfile, EOF being true implies that EOLN is true as well.

If a file is closed, EOF returns **true**. After a RESET takes place, EOF is **false** for the RESET file. If EOF becomes **true** during a GET or a READ, the data obtained is not valid.

8.4.4 ISNIN, ISINF, ISNUM

The three predicates ISNIN, ISINF, and ISNUM take either a **real** or **double** parameter and return **true** if that value is Not A Number, is an INFINITY value, or is a NUMBER, respectively. Otherwise they return **false**.

8.5 Value Conversion Functions

8.5.1 TRUNC – Truncate to Nearest Integer

The function TRUNC(x) truncates its argument "x" to the nearest integer. "x" must be of type **real** or **double**. If the result of truncating the argument "x" cannot be stored in an **integer** variable, the maximum negative **longint** value is returned.

For x >= 0, the result is the largest integer <= x.

For x < 0, the result is the smallest integer >= x.

8.5.2 ROUND – Round to Nearest Integer

The function ROUND(x) rounds its argument "x" to the nearest integer. "x" must be of type **real** or **double**. The result is of type **integer**. If the result of rounding the argument "x" cannot be stored in an **integer** variable, the maximum negative **longint** value is returned.

For x >= 0, the result is TRUNC(x+0.5).

For x < 0, the result is TRUNC(x-0.5).

8.5.3 ORD – Convert Type to Integer Value

The function ORD(x) returns an **integer** which is the ordinal number of the argument "x" in the set of values defined by the type of "x". The argument "x" can be any non-floating point scalar. For example:

```
var
   one_letter : char:
   converted : integer;

begin
   one_letter := 'm';
   converted := ORD(one_letter);
```

At the end of this program fragment, the variable "converted" has the value 109, since that is the ordinal position of lower case 'm' in the ASCII character set.

8.5.4 ORD4 – Convert to Long Integer

The function ORD4(x) returns a **longint** which is the ordinal number of the argument "x". As for ORD, the argument "x" can be any non-floating point scalar.

8.5.5 CHR – Integer to Character Representation

The function CHR(x) converts its argument "x" to a character. The argument "x" must be an **integer**. The result type of CHR is the character whose ordinal number is "x". The argument must therefore lie in the range 0 .. 255 for CHR to return a valid result.

8.6 Other Standard Functions

8.6.1 SUCC – Determine Successor of Value

The function SUCC(x) accepts an argument which is any scalar type except **real** or **double**. The result of SUCC is the successor value of the argument, if such a successor value exists.

SUCC(x) is undefined if "x" does not have a successor value.

8.6.2 PRED – Determine Predecessor of Value

The function PRED(x) accepts an argument which is any scalar type except **real** or **double**. The result of PRED is the predecessor value of the argument, if such a predecessor value exists.

PRED(x) is undefined if "x" does not have a predecessor value.

8.7 Miscellaneous Low Level Routines

8.7.1 MOVELEFT and MOVERIGHT

MOVELEFT and MOVERIGHT transfer a number of bytes from a source to a destination. MOVELEFT starts at the leftmost byte in the source (the first byte), while MOVERIGHT starts at the rightmost byte in the source (the last byte). In all cases the source and destination strings can overlap, with the appropriate undesired results if the move is in the wrong direction. The format of MOVELEFT and MOVERIGHT is:

procedure MOVELEFT(**var** source, **var** destination, length);

procedure MOVERIGHT(**var** source, **var** destination, length);

source	is the place to move bytes from.
destination	is the place to move bytes to.
length	is an integer specifying the number of bytes to move.

"source" and "destination" can be any sort of type. If either "source" or "destination" is an array, the array can be subscripted. If either "source" or "destination" is a record, a field specification can be given.

For a MOVELEFT, the byte at "source" is moved to "destination" and so on until the byte at "source"+"length"−1 is moved to "destination"+"length"−1. For a MOVERIGHT, the move starts from the other end, so that the byte at "source"+"length"−1 is moved to "destination"+"length"−1 and so on until the byte at "source" is moved to "destination".

Neither MOVELEFT nor MOVERIGHT perform any range checking. They should therefore be used with a modicum of caution.

Example of MOVELEFT

The example shown below illustrates how MOVELEFT can be used to "zero fill" an array.

```
var
  manifold: array[1 .. 100] of -128 .. 127;

  manifold[1] := 0;     { place an initial zero }
  MOVELEFT(manifold[1], manifold[2], 99);
```

8.7.2 FILLCHAR — Fill A Storage Region With A Character

FILLCHAR is a procedure that replicates a byte throughout a region of storage. The procedure definition of FILLCHAR is:

```
procedure FILLCHAR(var address; integer count; char byte);
```

address — is the address of an arbitrary storage location in memory. Note that 'address' is a **var** parameter to FILLCHAR, so it may not be the address of a **packed** object.

count — is the number of times that the next parameter — 'byte' should be replicated.

byte — is a single character value which is replicated throughout the region of storage starting at 'address' and ending at 'address'+'count'−1.

Example of FILLCHAR

The example shown below illustrates how FILLCHAR can be used to "space fill" a print buffer

```
var
  printbuf: array[1 .. 256] of char;

  FILLCHAR(printbuf, 256, ' ');
```

8.7.3 SIZEOF — Determine Size of Data Element or Type

SIZEOF is a function that returns the number of bytes that a variable or type is allocated. The function definition of SIZEOF is:

```
function SIZEOF(identifier): integer;
```

where "identifier" is a variable name or a type identifier. The SIZEOF function

is particularly useful as a parameter to MOVELEFT or MOVERIGHT, or in performing unit input-output, where the number of bytes to transfer must be known.

8.7.4 POINTER – Convert Integer Expression to Pointer

POINTER converts an integer expression to a pointer value. The function definition of POINTER is:

```
function POINTER(expression): universal;
```

POINTER converts the "expression", which must be an integer expression, to a pointer value. The result type of POINTER is a "universal" pointer type that has the type of **nil**, which means that it may be assigned to any pointer variable.

8.8 Control Procedures

8.8.1 EXIT – Exit from Procedure

EXIT provides the means to "get out of" a procedure prematurely. EXIT finds especial use in recursive applications such as expression evaluators or tree-walking procedures. Its effect is to cause an immediate (and clean) return from a named procedure or function. The procedure definition of EXIT is:

```
EXIT(name);
```

where "name" is the name of the procedure or function to be exited.

If the "name" parameter is the name of a recursive procedure or function, the *most recent* activation of that procedure or function is terminated.

Files that are local to an EXIT'ed procedure or function are not implicitly closed upon exit – they must be closed explicitly before the EXIT statement.

If an EXIT statement is made inside a function before any assignment is made to the function identifier, the result of the function is undefined.

EXIT is exactly the same as a **goto** label at the end of the named procedure or function.

8.8.2 HALT – Terminate Program with Return Value

The HALT procedure terminates the currently executing program. HALT returns a value to the host operating system to indicate a successful termination or an error termination. The procedure definition of the HALT procedure is:

```
HALT(i: integer);
```

The "i" parameter is optional. If the "i" parameter is omitted, by simply executing a

```
HALT
```

statement, the correct "no error" code is returned to the host operating system.

The HALT procedure also returns a value to the CALL function, described below.

A list of the values which the HALT procedure can return can be found in Appendix A – "Messages from the Pascal System".

8.8.3 CALL – Call up Another Program

The CALL function requests the host operating system to execute another program. The function definition of CALL is:

```
function CALL(pathname: stringtype;
              var infile, outfile: interactive | text;
              fargv: ?; fargc: integer): integer;
```

The parameters to the CALL function are:

pathname	is a **string** containing the pathname of the file in which the program resides which is to be run. The definition of what constitutes a pathname is operating system dependent.
infile and outfile	specify the standard input and standard output for the program specified by "pathname". In addition, the definition specifies whether the standard input and standard output files for that program are text files residing in the file system, or the user's terminal.
fargv	is an array of pointers to **strings** consisting of the options and filename arguments for the program in question.
fargc	is an **integer** count of the number of arguments in "fargv".

The value returned from the CALL function is either the value which a program returns via a HALT call, or is one of the operating system error codes.

NOTE: The CALL function is not available under all operating systems. See Appendix G for specifics about this operating system.

Chapter 9 – Pascal Compile Time Options

Pascal compile-time options are introduced via *toggles* embedded in comments. Comment toggle format is like:

```
(*$T params*)
    or
{$T params}
```

where either the (* and *) form, or the { and } form of comment delimiters may be used.

The toggle must immediately follow the opening comment delimiter, with no intervening spaces.

A comment toggle is always introduced by a $ sign. The $ sign is followed by the toggle letter, either in upper or lower case, followed by the parameters for that toggle. Compiler options that are followed by a + or – may be given in a list:

```
{$C+,I+,L- ....}
```

There must not be any spaces after the commas in the list. Scanning of a list of compiler options terminates if any incorrect syntax is encountered.

Compiler options do not obey any of the Pascal scope rules. Once an option is selected by a toggle, it remains in effect until another toggle in the source text de-selects that option. Compiler options are described in the list below.

Some of the descriptions of the compiler options make references to the options specified on the compiler command line. A description of the command line options can be found in Appendix G – "Using The Pascal Compiler".

$C+ or $C– Turns *C*ode generation on (+) or off (–). This is done on a procedure by procedure basis. The value of the options at the end of a procedure controls code generation. The default is C+.

$E filename starts listing *E*rrors to the file specified by "filename". Also see the $L option below.

$N+ or $N− Check the result of floating point expressions for validity. If this option is enabled, then the VALUE of most floating point expressions are checked for the values Not A Number and INFINITY. If present, a run time error is caused. The default is off.

$I filename *I*nclude the file specified by "filename" at this point in the source.

$I+ or $I− Turn automatic Input Output checks on (+) or off (−). The default is I+.

$L filename Make a compilation *L*isting on the file specified by "filename". If a listing file already exists, that file is closed and saved before the new file is opened.

$L+ or $L− Turn *L*isting on (+) or off (−) without changing the listing file name. The listing filename must be specified before turning listing on. The default is $L+ (listing on) when a listing file has been specified on the compiler's command line or $L− (listing off) when a listing file was not specified. When the list option is on, the listing is directed to whatever list file was specified on the Pascal compiler's command line.

$M+ or $M− The $M+ option specifies that the Pascal run-time system should check the stack and heap for overflow upon entry to each procedure. The $M+ option enables the check. The $M− option disables the check. The default setting is $M− (disable the check).

$F+ or$F− Generates code to use floating point hardware (+) or software (−). In those implementations without floating point hardware, this option is ignored. The default is off (i.e. use software).

$P+ or $P− Specifies whether the Pascal compiler should prompt the user for corrective action when errors are detected. The $P+ option indicates that the compiler should prompt the user as to whether to continue the compilation when errors are detected. The $P− option disables the prompting feature. This feature is also available via the **−p** or **+p** option on the compiler command line. The default setting of the $P option is operating system dependent.

$Q+ or $Q− Controls the amount of messages that the Pascal compiler prints while compiling a program. The $Q+ option results in fewer messages. The $Q− option results in more messages. The default setting of the $Q option depends upon the operating system on which the Pascal system is running.

$R+ or $R− Turns run-time *R*ange checking on (+) or off (−). At present, range checking is done in assignment statements, on array indexes, and for string value parameters. The default setting is $R+. Range checking is only done for user defined subrange, and SCALERS and for array indexing. The type **integer** is *NEVER* range checked. Compile-time range checking (i.e. v:=constant) for user defined types is always enforced.

$S segment Places code modules into the *S*egment specified by "segment" The default segment name is ' ' (eight spaces), which is where the main program and all built-in support code is always linked. All other code can be placed into any segment. Under most operating systems segmentation is automatically done by the system and there is no reason to explicitly segment programs (See Appendix G for more specific information if segmentation is meaningful in your environment).

$S+ or $S− Is the swapping option. The $S+ option specifies that the compiler should run in swapping mode. The $S− option specifies that the compiler should not run in swapping mode. In operating systems which support generalized overlay schemes, swapping mode means that the compiler runs in less memory, at the expense of a considerable speed penalty. The $S+ option (if used) must appear before the initial **program** or **unit** header, else the option has no effect. The default setting is $S− (do not run in swapping mode).

$U filename. Searches for subsequent *U*nits in the file specified by "filename".

$%+ or $%− Specifies that the percent sign % is a valid character (+) or is not a valid character (−) in identifiers. The default is $%−.

Appendix A – Messages from the Pascal System

This appendix describes the error messages that the Pascal system generates.

A.1 Compile Time Lexical Errors

10 Too many digits
11 Digit expected after '.' in a real number
12 Integer Overflow
13 Digit expected in the exponent of a real number
14 End of line encountered in a string constant
15 Invalid character in input
16 Premature end of file in source program
17 Extra characters encountered after the end of the program
18 End of file encountered in a comment

A.2 Compile Time Syntactic Errors

20 Illegal symbol
21 Error in simple type
22 Error in declaration part
23 Error in parameter list of a procedure or function
24 Error in constant
25 Error in type
26 Error in field list of a record declaration
27 Error in factor of an expression
28 Error in variable
29 Identifier expected

30 Integer expected
31 '(' expected
32 ')' expected
33 '[' expected
34 ']' expected
35 ':' expected

36 ';' expected
37 '=' expected
38 ',' expected
39 '*' expected

40 ':=' expected
41 **program** keyword expected
42 **of** keyword expected
43 **begin** keyword expected
44 **end** keyword expected
45 **then** keyword expected
46 **until** keyword expected
47 **do** keyword expected
48 **to** or **downto** keyword expected

50 **if** keyword expected
51 '.' expected
52 **implementation** keyword expected
53 **interface** keyword expected

A.3 Compile Time Semantic Errors

100 Identifier declared twice in the same block
101 Identifier is not of the appropriate class
102 Identifier not declared
103 Sign not allowed
104 Number expected
105 Lower bound exceeds upper bound
106 Incompatible subrange types
107 Type of constant must be integer
108 Type must not be real
109 Tagfield must be a scalar or subrange

110 Type incompatible with tagfield type
111 Index type must not be real
112 Index type must be scalar or subrange
113 Index type must not be **integer** or **longint**
114 Unsatisfied forward reference
115 Forward reference type identifier cannot appear in a variable declaration
116 Forward declaration – repetition of parameter list not allowed
117 Forward declared function – repetition of result type not allowed
118 Function result type must be scalar, subrange, or pointer
119 File is not allowed as a value parameter

120 Missing result type in function declaration
121 F-format for **real** type only
122 Error in type of parameter to a standard function
123 Error in type of parameter to a standard procedure

124 Number of actual parameters does not agree with declaration
125 Illegal parameter substitution
126 Result type of parametric function does not agree with declaration
127 Expression is not of **set** type
128 Only tests for equality allowed
129 Strict inclusion not allowed

130 Comparison of **file** variables not allowed
131 Illegal type of operand(s)
132 Operand type must be **boolean**
133 Set element type must be scalar or subrange
134 Set element types not compatible
135 Type of variable is not **array** or **string**
136 Index type is not compatible with declaration
137 Type of variable is not **record**
138 Type of variable must be **file** or pointer
139 Illegal type of loop control variable

140 Illegal Expression type
141 Assignment of files not allowed
142 Case selector incompatible with selecting expression
143 Subrange bounds must be scalar
144 Operand type conflict
145 Assignment to standard function is not allowed
146 Assignment to formal function is not allowed
147 No such field in this record
148 Type error in **read**
149 Actual parameter must be a variable

150 Multiply defined case selector
151 Missing corresponding variant declaration
152 **real** or **string** tagfields not allowed in variant record
153 Previous declaration was not **forward**
154 Substitution of standard procedure or function not allowed
155 Multiple defined label
156 Multiple declared label
157 Undefined label
158 Undeclared label
159 Value parameter expected

160 Multiple defined record variant
161 **File** not allowed here
162 Unknown compiler directive (not **external** or **forward**)
163 Variable cannot be a packed field
164 **Set of real** is not allowed
165 A field of a **packed record** cannot be a **var** parameter
166 Case selector expression must be a scalar or a subrange
167 String sizes must be equal

168 String too long
169 Value out of range

170 Cannot take the address of a standard procedure or function
171 Assignment to function result must be done inside that function
172 Control variable of a **for** statement must be local
173 BUFFERED or UNBUFFERED expected
174 NORMAL, LOCK, PURGE, or CRUNCH expected
175 **File** variable expected
176 Must be within the **procedure** or **function** being **exit**ed
177 Cannot pass **cexternal** as **procedure** or **function** parameter
178 Label value must be 0 to 9999

190 No such **unit** in this file

A.4 Specific Limitations of the Compiler

300 Too many nested record scopes
301 Set limits out of range (maximum sized set is 0 .. 2031)
302 String limits out of range
303 Too many nested procedures or functions
304 Too many nested **include** or **uses** files
305 **Include** not allowed in **interface** section
306 **Pack** and **unpack** are not implemented
307 Too many units
308 Set constant out of range
309 Maximum comparable **packed array of char** is of size 255 characters

310 Too many nested **with** statements
311 Too many nested **function** references
312 Record too big (maximum size is 32766 bytes)
313 Too many elements in an array (maximum size or elements is 32766)
314 Too many variables in one scope (maximum is 32766 bytes)

350 Procedure too large
351 File name in option too long

A.5 Input Output Errors

400 Not enough room for code file
401 Error in rereading code file
402 Error in reopening text file
403 Unable to open **uses** file
404 Error in reading **uses** file
405 Error in opening **include** file
406 Error in rereading previously read text block
407 Not enough room for intermediate code file

408 Error in writing code file
409 Error in reading intermediate code file

410 Unable to open listing file

A.6 Code Generation Errors

1000+ Code generator errors – in theory should never happen

Normally these errors indicate that an erroneous .I file has been specified as the input file to the code generator.

A.7 IORESULT Error Codes

The codes listed below are those that the IORESULT function returns.

0 No Error – indicates a good result
1 Parity error or CRC error
2 Invalid device number
3 Invalid input-output request
4 Nebulous Hardware Error
5 Volume went off-line
6 File lost in directory
7 Bad file name
8 No room on volume
9 Volume not found

10 File not found

11 Duplicate directory entry
12 File already open
13 File not open
14 Bad input information
15 Ring buffer overflow
16 Write protect
17 Invalid seek 64 Device error of unknown origin

Appendix B – Pascal Language Summary

.... "what *is* the use of repeating all that stuff", said the Mock Turtle, "if you don't explain it as you go along. It's by far the most confusing thing I ever heard!" ....

Lewis Carroll. Through the Looking Glass

B.1 Predefined Identifiers

Constants

maxint	TRUE	FALSE	

Types

Boolean	**interactive**	**longint**	**text**
char	**integer**	**real**	**double**

Variables

Argc	Argv	Input	Output
Stderr			

Procedures

CLOSE	HALT	PUT	UNITCLEAR
DELETE	INSERT	READ	UNITREAD
DISPOSE	MARK	READLN	UNITSTATUS
EXIT	MOVELEFT	RELEASE	UNITWRITE
FILLCHAR	MOVERIGHT	RESET	WRITE
GET	NEW	REWRITE	WRITELN
GOTOXY	PAGE	SEEK	

Functions

ABS	EOLN	MEMAVAIL	SCANEQ
ARCTAN	EXP	ODD	SCANNE
BLOCKREAD	FILLCHAR	ORD	SIN
BLOCKWRITE	IORESULT	ORD4	SIZEOF
CHR	ISINF	POINTER	SQR
CONCAT	ISNAN	POS	SQRT
COPY	ISNUM	PRED	SUCC
COS	LENGTH	PWROFTEN	TRUNC
EOF	LN	ROUND	UNITBUSY

B.2 Pascal Syntax Definitions

Syntactic constructs enclosed between "angle brackets" < and > define the basic language elements. Every language construct should eventually be defined in terms of basic lexical constructs defined in the remainder of this appendix.

A construct appearing outside the angle brackets stands for itself, that is, it is supposed to be self denoting. Such a construct is known as a *terminal symbol.* Terminal symbols and reserved words appear in **bold face text** throughout this manual.

The symbol ::= is to be read "defined as".

The symbol .. means "through", indicating an ordered sequence of things where only the start and end elements are specified. (The reader is left to infer the middle elements). For example, the notation 'a' .. 'z' means "the ordered collection starting with the letter 'a', ending with the letter 'z', and containing the letters 'b', 'c'....'x', 'y' in between". In other words, all the lower case letters.

The "vertical bar" symbol | is read as "or". It separates sequences of elements that represent a choice of one out of many.

The metalanguage construct {...} (elements inside braces) enclose elements which are to be repeated "zero to many times". Although the braces are also used as one of the forms of comment delimiters in Pascal, this should not cause any ambiguity. The one case where ambiguity would occur is in the definition of comments, and this is explicitly pointed out at that time.

The Pascal compiler recognizes the following *alphabet* or *character set*.

```
        <letter>          ::= 'A' .. 'Z', 'a' .. 'z', and '_'

        <digit>           ::= '0' .. '9'

      <hex digit>         ::= <digit> | 'a' .. 'f' | 'A' .. 'F'

<ASCII graphic characters> ::= ! " # $ % & ' ( ) * =
                               + - , . / < > ? _ \ [
                               @ ^ | ~ ` { } ; : ]

<identifier> ::= <letter> { <letter> | <digit> }

<unsigned integer> ::= <digit> {<digit>}
  <unsigned real> ::=
          <unsigned integer>.<unsigned integer>
        | <unsigned integer>.<unsigned integer>E<scale factor>
        | <unsigned integer>E<scale factor>
        | <unsigned integer>.<unsigned integer>D<scale factor>
        | <unsigned integer>D<scale factor>

<unsigned number>::= <unsigned integer> | <unsigned real>

  <scale factor>    ::= <unsigned integer> | <sign><unsigned integer>

       <sign>       ::= + | -

<hex number> ::= $<hex digit> {<hex digit>}

        <string>    ::= '<character> {<character>}'

<character value>   ::= \<two digit hexadecimal number>

<label> ::= <unsigned integer>

<comment>     ::={ <any printable characters except "}"> }
                | (* <any printable characters except "*)" *)

<any printable character> includes carriage-return, line-feed,
                          tab, and so on.

<constant identifier> ::= <identifier>

      <constant> ::= <unsigned number>
                   | <sign> <unsigned number>
                   |    <constant identifier>
                   |      <sign> <constant identifier>
                   |        <string>

<constant definition> ::= <identifier> = <constant>

<type declaration>    ::=  type <type spec> {;<type spec>}

    <type spec> ::= <type identifier> = <Pascal type>
```

```
<simple type>   ::=  <scalar type>
                   | <standard type>
                   |   <subrange type>
                   |     <type identifier>

<scalar type>   ::=  (<identifier> {,<identifier>})

<subrange type>  ::=
                <subrange type identifier> | <lower> .. <upper>

          <lower>::= <signed scalar constant>
          <upper>::= <signed scalar constant>
     <structured type> ::= <unpacked structured type>
                         | packed <unpacked structured type>

<unpacked structured type> ::= <array type>
                             | <string type>
                             |   <record type>
                             |     <set type>
                             |       <file type>

  <array type>      ::=  array [<index list>] of <type>

  <index list>      ::=  <simple type> {, <simple type>}

<component type>  ::=  <type>

 <string type>      ::=  string[<static length>]

<static length>     ::=  integer constant in the range 1 .. 255

<record type> ::= record <field list> end;
<field list>  ::= <fixed part>
                | <fixed part> ; <variant part>
                |   <variant part>

<fixed part> ::= <record section> {; <record section>}
<record section> ::= <field identifier list> : <type>
<field identifier list> ::= <field identifier> {,<field identifier>}

<variant part> ::=
        case {<tag field>} <type identifier> of <variant list>
<variant list> ::= <variant> {; <variant>}
<variant> ::= <case label list> : (<field list>)
<case label list> ::= <case label> {, <case label>}
<case label> ::= <constant>
<tag field> ::= <identifier>:

 <set type> ::= set of <simple type>
```

```
<file type> ::= file of <type>
              |  file
<pointer type> ::= ^<type identifier>
<variable declaration> ::=
          <identifier> {,<identifier>}: <data type>;

<variable> ::= <entire variable>
             | <component variable>
             |   <referenced variable>

<entire variable> ::= <variable identifier>

<component variable> ::= <indexed variable>
                       |   <field designator>
                       |     <file buffer>

<indexed variable> ::= <array variable> <subscript list>

 <subscript list>   ::= [ <expression> {,<expression>} ]
                      | [ <expression> ] {[ <expression> ]

<field designator>  ::= <record variable>.<field identifier>

 <file buffer> ::= <file variable>^
<file variable> ::= <variable>

<referenced variable> ::= <pointer variable>^

 <pointer variable>  ::= <variable>
      <unsigned constant>::= <unsigned number>
                           |      <string>
                           |        <constant identifier>
                           |          nil

      <factor>  ::= <variable>
                  |   <unsigned constant>
                  |     <function designator>
                  |       <set constructor>
                  |         (<expression>)
                  |           not <factor>

 <set constructor> ::= [ <element> {,<element>} ]
     <element> ::= <expression>
                 |   <expression> .. <expression>

       <term>  ::= <factor>
                 |   <term> <multiplying operator> <factor>

   <simple expr>   ::= <term>
                 |   <simple expr> <adding operator> <term>
                 |     <adding operator> <term>
```

```
<expression> ::=
    <simple expr>
        |  <simple expr> <relational operator> <simple expr>

<multiplying operator> ::= * | / | div | mod | and

<adding operator> ::= + | - | or

<sign operator> ::= + | -

<relational operator> ::= = | <> | > | < | >= | <= | in

<assignment statement> ::=
        <variable> := <expression>
    |   <function identifier> := <expression>

<procedure call statement> ::=
        <procedure identifier> <actual parameter list>
    |   <procedure identifier>

<actual parameter list> ::=
        (<actual parameter> {,<actual parameter>})

<actual parameter> ::= <expression>
        |  <procedure identifier>
        |  <function identifier>

<structured statement> ::= <begin statement>
        |  <if statement>
        |  <while statement>
        |  <repeat statement>
        |  <for statement>
        |  <case statement>

<begin statement> ::= begin <statement list> end

<statement list> ::= <statement> {; <statement>}

<if statement> ::=
    if <Boolean expression> then <statement>
    |  if <Boolean expression> then <statement> else <statement>

<case statement> ::= case <expression> of <cases>
        {otherwise: <statement>} end

    <cases> ::= <a case> {; <a case>}
    <a case> ::=
        <selection spec> {, <selection spec>} : <statement>

<selection spec> ::= <simple constant scalar expression>

<while statement> ::= while <expression> do <statement>

<repeat statement> ::= repeat <statement> until <expression>
```

```
<for statement> ::=
              for <control variable> := <for list> do <statement>
<for list> ::= <initial value> to <final value>
             | <initial value> downto <final value>

<control variable> ::= <identifier>

  <initial value>  ::= <expression>
   <final value>   ::= <expression>

<with statement> ::=
                  with <record variable> {,<record variable>}
                                                  do <statement>

<goto statement> ::= goto <label>

<unit> ::= unit <identifier>;
              <interface part>
              <implementation part>
           end.
<interface part> ::= interface
                       <uses clause>
                         <constant definition part>
                           <type definition part>
                             <variable definition part>
                               <procedure and function declaration part>
<implementation part> ::= implementation
          <label declaration part>
            <constant definition part>
              <type definition part>
                <variable definition part>
                  <procedure and function declaration part>

<uses clause> ::= uses <unit name> {,<unit name>}

<declaration> ::= <constant declaration>
                | <type declaration>
                |   <variable declaration>
                |     <procedure or function declaration>

<program> ::= <program heading> <block>.

<program heading> ::=
          program <identifier> (<program parameters>);

<program parameters> ::= <identifier> {,<identifier>}

<label declaration part> ::= label <label> {, <label>};

<constant definition part> ::= const <constant definition list>;
```

```
<constant definition list> ::=
                    <constant definition> {; <constant definition>}

<type definition part> ::= type <type definition list>;

<type definition list> ::= <type definition> {; <type definition>}

<variable declaration part> ::= var <variable declaration list>

<variable declaration list> ::=
                <variable declaration> {;<variable declaration>}

<procedure declaration> ::= <procedure heading> <block>
 <function declaration> ::= <function heading> <block>
<block> ::= <label definition part>
                <constant definition part>
                  <type definition part>
                    <variable declaration part>
                      <procedure and function declaration part>
                        <statement part>

<statement part> ::= begin <statement list> end

<statement list> ::= <statement> {; <statement>}

<procedure heading> ::=
                procedure <identifier>; {<attribute>;}
            |     procedure <identifier> (<formal parameters>);
                                {<attribute>;}

 <function heading> ::=
function <identifier>:<result type>; {<attribute>;}
| function <identifier> (<formal parameters>):
                                <result type>; {<attribute>;}

<formal parameters> ::= <formal parameter> {;<formal parameter>}

<formal parameter> ::= <parameter group>
                     |     var <parameter group>
                     |       <procedure heading>
                     |         <function heading>

<parameter group> ::=
                <identifier> {,<identifier>}:<type identifier>

<attribute> ::= external | forward | cexternal

<result type> ::= <simple type>
```

Appendix C – Relationships to ISO Pascal

Myself when young did eagerly frequent
Doctor and Saint, and heard great argument
About it and about: but evermore
Came out by the same door as in I went.
..... Omar Khayam, The Rubaiyat

The International Standards Organization (ISO) and the American National Standards Institute (ANSI) are engaged in a joint effort to define a Pascal Standard.

In general, SVS Pascal conforms to the (proposed) ISO Pascal standard as defined in Pascal User Group News, Number 20, December 1980. There are however some differences that are spelled out here.

In SVS Pascal, thirty-one characters are significant in identifiers. Linkable external names have only eight significant characters.

The Pascal standard procedures PACK and UNPACK are not supplied.

Conformant arrays are not implemented in accordance with the level 0 (U.S) standard.

There is a small difference in the way that a text file is handled if the text file is associated with an interactive terminal.

There is a **string** basic data type implemented.

There is a **double** basic data type implemented. The **double** data type is a double precision **real** data type.

There is an **otherwise** clause in the **case** statement. This provides for a "what to do if the case selector matches none of the cases". Standard Pascal considers this situation an error.

SVS Pascal implements a **longint** data type, which occupies four bytes instead of the two bytes of the standard **integer** data type.

The **and**, **or**, and **not** operators can be applied to operands of type **integer** as well as operands of type **Boolean**. When applied to operands of type **integer**, these operators perform bitwise logical and, logical or, and logical not operations on their operands.

SVS Pascal supports many extensions. These mainly derive from the UCSD P-System.

Appendix D – Relationships to UCSD Pascal

The University of California at San Diego (UCSD) implemented a widely used Pascal system, oriented towards small, personal computer systems. This implementation is known as UCSD Pascal.

SVS Pascal uses a number of ideas from UCSD Pascal. The main areas where SVS Pascal conforms to UCSD Pascal are:

1. Independent compilation is supported through the **unit** concept of UCSD Pascal. The **interface**, **implementation** and **uses** statements are implemented.
2. There is an *include* capability.
3. Many of the UCSD Pascal compatible standard procedures and functions are implemented the same as UCSD Pascal.

D.1 Differences from UCSD Pascal

In SVS Pascal, the underline character _ is significant in identifiers. In UCSD Pascal it is ignored so that the identifiers "Space_Out" and "SpaceOut" are identical. In SVS Pascal they are considered two different identifiers.

SVS Pascal supports a long integer type, with the predefined type name **longint**. The UCSD construct **integer**[nn] is not implemented.

There is a **double** basic data type implemented. The **double** data type is a double precision **real** data type.

Fields of **packed records** and elements of **packed arrays** can never be passed as reference parameters to procedures, even in those places where UCSD Pascal allows.

The SVS Pascal string type **packed array**[low..high] **of char** must have a lower bound of 1 to be compatible with literal strings, or to be used in array comparisons. UCSD Pascal allows any lower bound.

SVS Pascal does not have the reserved word **segment**. Consequently there is no **segment procedure** or **segment function**. To segment a SVS Pascal

program, use the $S compiler option, which directs the compiler to place generated object code in a named segment. See Chapter 9 which contains a section on compiler options.

SVS Pascal does not implement **unit** initialization code.

SVS Pascal does not supply special units such as APPLESTUFF or TURTLEGRAPHICS.

SVS Pascal does not have any default **string** length. Instead of the declaration

```
var x: string;
```

use the declaration

```
var x: string[80];
```

SVS Pascal does not have a predefined file called "keyboard".

SVS Pascal implements sets with elements 0 through 2031, whereas UCSD Pascal implements 0 through 511.

Packing algorithms for **arrays** and **records** are different.

Internal storage for **sets** is different.

SVS Pascal does not support comparison of **arrays** and **records**, with the single exception that **packed array**[1..n] **of char** can be compared.

Predefined **string** procedures and functions must have **string** variable or string literal parameters. That is, not **packed array of char** or **char** variable parameters.

SVS Pascal does not implement the procedure STR, since there is no **integer**[nn] type.

The file procedures RESET and REWRITE require two parameters, namely (file,string).

End-of-file character from the keyboard is Control-D instead of Control-C.

SVS Pascal **text** files must be declared as **packed file of char**.

SVS Pascal text file reads allow additional parameters of **packed array of char**.

SVS Pascal text file writes allow additional parameters of **packed array of char** and **Boolean**.

Under most operating systems, SVS Pascal does not implement the unit I/O routines such as UNITREAD, UNITWRITE, and UNITWAIT.

SVS Pascal does not implement TREESEARCH.

SVS Pascal limits the EXIT procedure to exiting statically compiled procedures or functions or the main-program. The argument to EXIT must be the name of the routine to exit. That is, EXIT(PROGRAM) is not allowed.

The MEMAVAIL procedure returns the number of *bytes* of available memory. The return parameter is of the type **longint**. See the Section on "Memory Management". Under some operating systems MEMAVAIL is not meaningful.

SVS Pascal implements two procedures SCANEQ (scan equal) and SCANNE (scan not equal), whereas UCSD Pascal implements a single SCAN procedure with a = or <> parameter.

SVS Pascal does not have any INTRINSIC units.

SVS Pascal does not implement the **unit** initialization section in units.

SVS Pascal implements an optional **otherwise** clause in **case** statements. If the **otherwise** clause is present, it must be the last statement. For example:

```
case huh of
   1: do_this;
   3,5: do_that;
otherwise:
   do_the_other;
end;
```

SVS Pascal implements true global **goto** statements. The UCSD Pascal {$G+} compiler option is not needed in order to use **goto** statements.

SVS Pascal has predeclared variables ARGC and ARGV that describe the number and value of any parameters passed from the command line to a running program.

Procedures and functions may be passed as parameters. The implementation is consistent with the proposed ISO standard Pascal.

ORD(**Boolean** Expression) works properly in SVS Pascal.

The **mod** operator works properly in SVS Pascal.

SVS Pascal has added the unary operator @, which stands for "address of". Placing the @ in front of a variable, function, or procedure, generates the address of that entity. The type returned is the type of **nil**, that is, it can be assigned to any pointer variable. The @ operator does not work with most of the predefined procedures and functions such as ORD or READLN.

SVS Pascal has added the function ORD4. It is the same as ORD except that it returns a 32-bit integer.

All **integer** arithmetic operations are done at a precision of either 16 or 32 bits, depending on the maximum size of any arguments. The rules are similar to FORTRAN's single and double precision reals.

SVS Pascal statement labels are restricted to the range 0 through 9999, as in the ISO Pascal standard.

SVS Pascal provides for hexadecimal integer constants. A hexadecimal constant is prefixed with a $ sign. Hexadecimal numbers must be 32 bits long to be considered signed numbers, that is, $FFFF represents 65536, not −1. To represent −1, code the hexadecimal constant $FFFFFFFF.

The **and**, **or**, and **not** operators can be applied to operands of type **integer** as well as operands of type **Boolean**. When applied to operands of type **integer**, these operators perform bitwise logical and, logical or, and logical not operations on their operands.

Appendix E – Data Representations

This appendix describes the ways that SVS Pascal represents data in storage, how that data is packed for data objects that have the **packed** storage attribute, and the mechanisms for passing parameters to procedures and functions. This appendix is intended as a guide to those programmers who wish to write modules in languages other than Pascal and have those modules interface to Pascal.

E.1 Storage Allocation

This section describes the way in which storage is allocated to variables of various types. The storage allocation described here is for unpacked items.

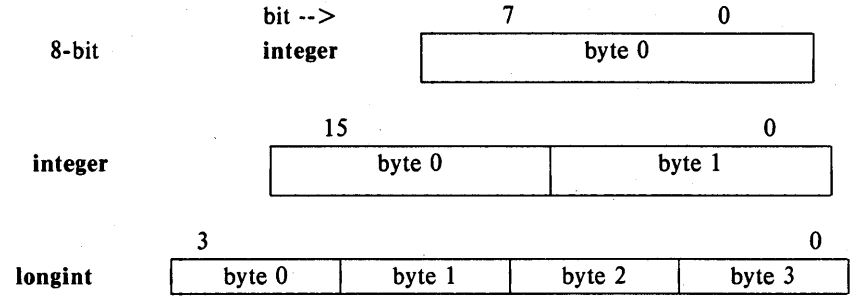
In general, any *word* value is always aligned on a word boundary. Anything larger than a word is also aligned on a word boundary. Values that can fit into a single byte are aligned on a byte boundary.

A **Boolean** variable	occupies one byte of storage, aligned on a byte boundary. A value of 0 represents the value **false**. A value of 1 represents the value **true**. Any other value is an "undefined" **Boolean** value.
A scalar (ordinal) type	of 128 elements or less, occupies one byte of storage, aligned on a byte boundary. If there are more than 128 elements in the scalar types, it then occupies a word. Scalar types are assigned the values 0, 1, 2, ..., n−1, where "n" is the cardinal number of elements in the scalar.
Subrange elements	in the range −128 .. 127 occupy one byte, aligned on a byte boundary. A subrange element in the range −32768 .. 32767 occupies one word, aligned on a word boundary. A subrange element greater than that occupies two words, aligned on a word boundary.

An unpacked **char** element	is considered to be a subrange of 0 .. 255. This means that it occupies a word.
An **integer** element	occupies one word, aligned on a word boundary.
A **longint** element	occupies two words, aligned on a word boundary.
real elements	occupy two words, aligned on a word boundary. A **real** element has a sign bit, an 8-bit exponent and a 24-bit mantissa. SVS Pascal **real** elements conform to the IEEE standard for reals as defined in the March 1981 Computer magazine. The layout of a **real** element is shown below. The range of **real** numbers is approximately −3.4E38 .. +3.4E38, with a precision of approximately seven decimal places. Normal arithmetic operations upon **real** data types can result in the "extreme values" of plus infinity, minus infinity, or Not a Number (NaN). These are described below.
double elements	occupy four words, aligned on a word boundary. A **double** element has a sign bit, an 11-bit exponent and a 53-bit mantissa. SVS Pascal **double** elements conform to the IEEE standard for double precision as defined in the March 1981 Computer magazine. The layout of a **double** element is shown below. The range of **double** numbers is approximately −1.0D308 .. +1.0D308, with a precision of approximately 15 decimal places. Normal arithmetic operations upon **double** data types can result in the "extreme values" of plus infinity, minus infinity, or Not a Number (NaN). These are described below.

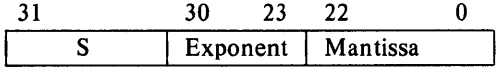
Whatever the size of the data element in question, the most significant bit of the data element is always in the lowest numbered byte of however many bytes are required to represent that object. The diagrams below should clarify this.

E.2 Representation of Integers



E.3 Representation of Reals and Doubles

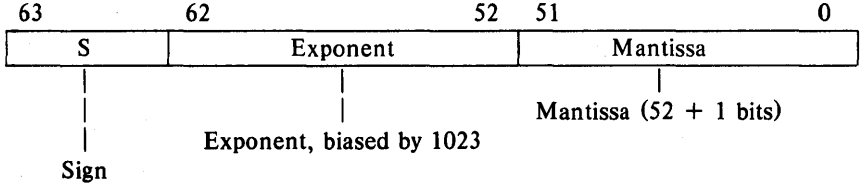
real and **double** data elements are represented according to the proposed IEEE standard as defined in Computer magazine of March, 1981. The diagrams below illustrate the representation.



real *Data Representation*

The format for a **real** or single-precision floating-point number is as shown above. The three field of a **real** are as follows:

- a one-bit sign bit designated by "S" in the diagram above. The sign bit is a **1** if, and only if, the number is negative.
- an 8-bit biased exponent. The values of all ones and all zeros are reserved values for the exponent.
- a 24-bit mantissa, with the high order 1 bit "hidden".



double *Data Representation*

The parts of **double** numbers are as follows:

- a one-bit sign bit designated by "S" in the diagrams above. The sign bit is a **1** if, and only if, the number is negative.
- an 11 bit biased exponent. The values of all zeros and all ones are a one-bit sign bit designated by "S" in the diagrams above.
- a normalized 53-bit mantissa, with the high-order **1** bit "hidden".

A **real** or **double** number is represented by the form:

$$2^{\text{exponent-bias}} * 1.f$$

where 'f' is the bits in the mantissa.

Normalized **real** and **double** numbers are said to contain a "hidden" bit, providing for one more bit of precision than would normally be the case.

E.4 Representation of Extreme Numbers

When **real** or **double** data elements are stored in the system, there arises the question of how to represent "values" such as positive and negative infinity. The discussion below describes the representations of these extreme numbers, and their behavior in expression evaluation.

zero (signed) — is represented by an exponent of zero, and a mantissa of zero.

denormalized numbers — are a product of "gradual underflow". They are non-zero numbers with an exponent of zero. The form of a denormalized number is:

$$2^{\text{exponent-bias}+1} * 0.f$$

where 'f' is the bits in the mantissa.

signed infinity — (that is, affine infinity) is represented by the largest value that the exponent can assume (all ones), and a zero mantissa.

Not-a-Number (NaN) is represented by the largest value that the exponent can assume (all ones), and a non-zero mantissa. The sign is usually ignored.

Normalized **real** and **double** numbers are said to contain a "hidden" bit, providing for one more bit of precision than would normally be the case.

E.4.1 Hexadecimal Representation of Selected Numbers

Value	real	double
+0	00000000	0000000000000000
−0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
−1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000
+Infinity	7F800000	7FF0000000000000
−Infinity	FF800000	FFF0000000000000
NaN	7F8xxxxx	7FFxxxxxxxxxxxx

E.4.2 Deviations from the Proposed IEEE Standard

Deviations from the proposed IEEE standard in this implementation are as follows:

- affine mapping for infinities,
- normalizing mode for denormalized numbers,
- rounds approximately to nearest - 7 or more guard bits are computed, but the "sticky" bit is not,
- exception flags are not implemented,
- conversion between binary and decimal is not implemented.

E.4.3 Arithmetic Operations on Extreme Values

This subsection describes the results derived from applying the basic arithmetic operations on combinations of extreme values and ordinary values.

No traps or any other exception actions are taken.

All inputs are assumed to be positive. Overflow, underflow, and cancellation are assumed not to happen.

In all the tables below, the abbreviations have the following meanings:

Abbreviation	Meaning
Den	Denormalized Number
Num	Normalized Number
Inf	Infinity (positive or negative)
NaN	Not a Number
Uno	Unordered

Addition and Subtraction					
Left Operand	Right Operand				
	0	Den	Num	Inf	NaN
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Note 1	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Note 1: Inf + Inf = Inf; Inf − Inf = NaN

Multiplication					
Left Operand	Right Operand				
	0	Den	Num	Inf	NaN
0	0	0	0	NaN	NaN
Den	0	0	Num	Inf	NaN
Num	0	Num	Num	Inf	NaN
Inf	NaN	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Division					
Left Operand	**Right Operand**				
	0	Den	Num	Inf	NaN
0	NaN	0	0	0	NaN
Den	Inf	Num	Num	0	NaN
Num	Inf	Num	Num	0	NaN
Inf	Inf	Inf	Inf	Nan	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Comparison					
Left Operand	**Right Operand**				
	0	Den	Num	Inf	NaN
0	=	<	<	<	Uno
Den	>		<	<	Uno
Num	>	>		<	Uno
Inf	>	>	>		Uno
NaN	Uno	Uno	Uno	Uno	Uno

Notes:

NaN compared with NaN is Unordered, and also results in inequality.

+0 compares equal to −0.

Max					
Left Operand	**Right Operand**				
	0	Den	Num	Inf	NaN
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Min					
Left	**Right Operand**				
Operand	0	Den	Num	Inf	NaN
0	0	0	0	0	NaN
Den	0	Den	Den	Den	NaN
Num	0	Den	Num	Num	NaN
Inf	0	Den	Num	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

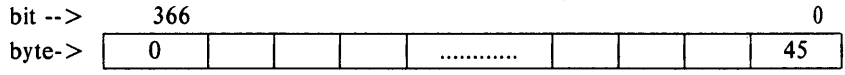
E.5 Representation of Sets

SVS Pascal represents a set like a "giant integer". The zeroth element of a set is always present in the set. Suppose that a type and a variable are defined as in this example.

```
type
   days_in_year = set of 1 .. 366;

var
   blarg: days_in_year;
```

The representation for the variable "blarg" is as in the diagram below:



The number of bytes required to contain this a set of 1 .. 366 is 366/8 which is 46 bytes. The storage is allocated accordingly as shown in the above diagram. The value 366 mod 8 is 6, and there is one unfilled bit in the least significant byte of the set.

E.6 Representation of Arrays

Components of unpacked arrays and records are allocated contiguously as defined above. There is no attempt made to conserve space in units smaller than bytes.

Arrays are stored in row order, that is, the last index varies fastest. This follows from the strict definition that a multi-dimensional array in Pascal is actually an:

```
array[first index] of array[second index] .....
        of array[n'th index] of whatever type;
```

E.6.1 Representation of Pointers

Pointers always occupy four bytes. The **nil** pointer is represented by a value of zero (0).

E.7 Packing Methods

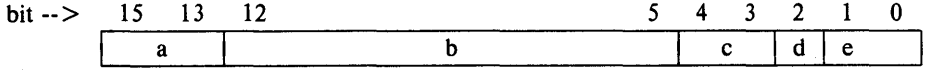
Packed records are expensive in terms of the amount of generated code needed to reference a field of a packed record. In general, avoid packing records unless there are many more instances of a particular records than there are references to it.

Components of a **packed record** are allocated in the order in which they appear. Components never cross word boundaries. The allocator never backtracks to fill in holes in the structure.

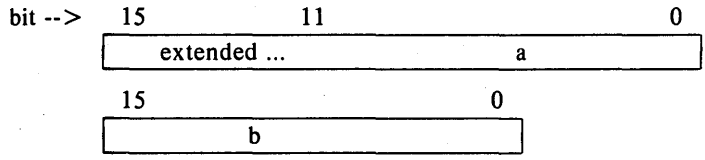
Within a word, components are allocated on bit boundaries. The allocation proceeds from the least significant end of the word towards the most significant end. If there is not enough room in the current word for a component, a hole is left in the current word, and the next word is started.

After allocation, it is possible that the allocator might shift and expand fields in a word to utilize what would otherwise be holes in the record. For example, a signed field might be expanded to use the remainder of a word for faster access, or two 5-bit fields might be allocated a full byte each. The diagrams below provide graphic illustrations of the packing methods. In each case, a **type** definition is given, followed by a diagram of how that type is allocated.

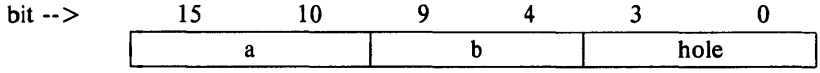
```
packed record
   a: 0 .. 7;
   b: char;
   c: 0 .. 3;
   d: Boolean;
   e:  0 .. 3;
end;
```



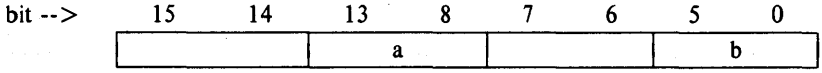

```
packed record
   a: 0 .. 4095;
   b: char;
end;
```



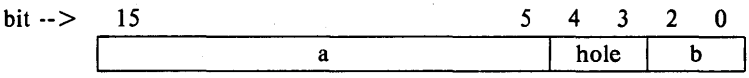

```
packed record
   a: 0 .. 63;
   b: 0 .. 63;
end;
```



The record above is allocated as in the above picture, but will be re-allocated as shown below.




```
packed record
   a: -1024 .. +1023;
   b: 0 .. 7;
end;
```



In the last example above, the signed subrange field was moved up to the left hand end of the word and sign extended for faster access.

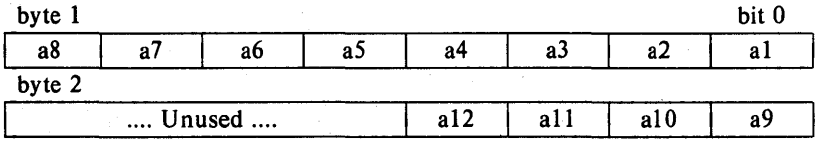
Packed arrays are also code consuming, with one exception: **packed array of char** is treated as a special case, and the generated code is compact.

Elements of **packed arrays** are stored with multiple values in a byte whenever more than one value can fit in a byte. Elements are allocated on 1, 2, 4 or 8-bit boundaries. This only happens when the value requires 4 bits or less. 3-bit values are stored in 4 bits.

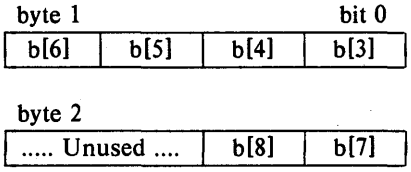
The first value in a **packed array** is stored in the lowest numbered bit position of the lowest addressed (that is, the most significant) byte. Subsequent values are stored in the next available higher numbered bit positions in that byte. When the first byte is full, the same positions are used

in the next higher addressed byte. Consider the following examples:

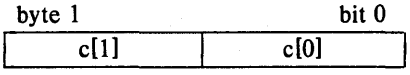
```
var
   a: packed array[1 .. 12] of boolean;
```



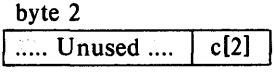

```
var
   b: packed array[3 ..8] of 0 .. 3;
```




```
var
   c: packed array[0 .. 2] of 0 .. 7;
                    or
   c: packed array[0 .. 2] of 0 .. 15;
```




```
var
   c: packed array[0 .. 2] of 0 .. 7;
                    or
   c: packed array[0 .. 2] of 0 .. 15;
```



E.8 Parameter Passing Mechanism

This Section describes the way in which parameters are passed in SVS Pascal.

Parameters are passed on the stack. Parameters are pushed onto the stack in order in which they are declared in **procedure** and **function** declarations.

If the callee is not a **procedure** or **function** at the global level, the static link is the last thing pushed onto the stack before the routine is called.

Upon return from a routine, all parameters are discarded from the stack. Nothing should be on the stack upon return.

var parameters (call by reference) always have a four-byte pointer to the variable pushed onto the stack.

Value parameters are divided into the three categories of **sets**, **doubles**, and everything else.

The caller always pushes **sets** onto the stack. A **set** which occupies one byte is pushed with a move.b instruction. A **set** which occupies more storage than one byte is pushed with the least significant element in the most significant word. Thus the representation of a **set** on the stack is the same as the representation in memory.

The caller always pushes **doubles** onto the stack as well. This is usually accomplished by two move.l instructions in such a manner that the representation a **double** on the stack is the same as the representation in memory (that is, with the sign bit in the lowest addressed byte).

Other value parameters are pushed as follows:

- a one-byte value is pushed with a move.b instruction.
- a two-byte value is pushed with a move.w instruction.
- a four-byte value is pushed with a move.l instruction.
- if a value is longer than four bytes, and not a **double**, the *address* of the data is pushed onto the stack and the called **procedure** or **function** copies the data into local storage.

Procedure and **function** parameters are pushed as follows:

- the address of the **procedure** or **function** is pushed onto the stack.
- the static link is then pushed onto the stack if the **procedure** or **function** is not at the global (outermost) level. If the **procedure** or **function** is global (at the outermost level), the value **nil**(0) is pushed onto the stack instead of the static link.

Function results are returned in register **D0**, or in the case of a **double function** in **D0** and **D1**.

E.9 Register Conventions

Registers **A0**, **A1**, **D0**, **D1**, and **D2** are available as scratch registers. That is, they may be clobbered by a **function** or **procedure**. All other registers must

be preserved across calls. In addition, register **A4** and **A5** must contain their original values whenever any external routine is called. **A4** is used in addressing external entry points and **A5** is used to access the standard input and output, argc and argv, ioresult, etc.

E.10 Limitations On Size of Variables

There is no limitation on the number of bytes allocatable for variables. However, a maximum of 30K bytes of *value* parameters cannot be exceeded. Furthermore, when more than 30K bytes of variables exist in either the main program's global scope, or in any local scope of a **procedure** or **function** (but not **unit** globals), the largest values will be accessed via a pointer, resulting in somewhat slower code. This mechanism is transparent to the user, so that no changes to source code are required.

Global variables in **units** are accessed via 32-bit absolute addressing modes. Therefore the pointer mechanism does not apply to **units** with more than 30K bytes of globals.

The maximum size of a **record** variable is 32K bytes.

There is no limitation on the size of variables which can be allocated by the NEW procedure.

E.11 Compiler Generated Linker Names

This section describes the manner in which the Pascal compiler generates names of local and global procedures so that the Linker can resolve external references at link time.

Procedures which are global (or external) are given the names which the user assigns to them. The compiler converts all such names to upper case, and truncates them to eight characters in length.

Procedures which are local (not visible in the global scope) are assigned names of the form:

```
$nnn
```

where 'nnn' is a decimal number. The numbers may possibly have trailing spaces. Procedures of the same name but in different scopes have different names. In other words, all local names in a given compilation unit are unique.

When the linker or librarian sees a collection of compiled units, the local names may be renumbered, but the actual name that the user assigned to the procedures are carried along with the number.

Appendix F – Operating the SVS Pascal System

This appendix will describe those characteristics of the SVS **Pascal** system which are similar among the various environments in which the system operates. The appendix which follows this one describes the implementation specific details of the **Pascal** system under your operating system. The information in this appendix describes the **Pascal** system in the form it is released by SVS. Some of the vendors of the system provide additional utilities which can used in conjunction with SVS **Pascal** and which may alter the appearance of the system.

F.1 System Components

In order to most effectively utilize the SVS **Pascal** system, it is necessary to understand the function and operation of its various components. In all environments a completely straight forward procedure is provided for compiling and executing simple **Pascal** programs (see next appendix). The information provided here, will only be necessary for more complicated situations involving separate compilation and multiple source languages.

F.1.1 Compiler Front End

Pascal source programs (actually **Pascal** compilation units) are accepted by the compiler front end, syntax checked, and an intermediate representation of the program is written to a file. This file is passed to the code generator which generates object code. The input source program may "include" other files (see Chapter 9). In addition to the input source file, the **Pascal** compiler front end accepts certain directives from the command line, which are described in the "Command Line Directives and Compiler Options" section of this appendix.

Input files to the **Pascal** front end generally are files with names which end in ".pas", although this differs among operating environments. The output file from the **Pascal** compiler front end is an intermediate representation of the program which is placed in a file which generally ends in ".i". There is virtually nothing which can be done with this ".i" file except provide it as input to the code generator.

F.1.2 Code Generator

The code generator for the **Pascal** system accepts as input the ".i" file produced by the front end and generates linkable object code in a file with a name which generally ends in ".obj".

The same code generator is utilized in compiling **SVS Pascal**, **SVS FORTRAN**, and **SVS C** and the resulting ".obj" files are linkable providing the applicable rules are followed.

F.1.3 Linker

A utility is provided with **SVS Pascal** for linking ".obj" files with each other and with run time libraries which are part of the language system. The linker is highly specific to the operating environment and its operation is described in detail in the following appendix. There is, however, certain general information which applies to all of the linkers.

Each linker accepts as inputs ".obj" files and produces an output which is acceptable to the operating system as an object file. In some operating environments, the linker's output file is further linkable in the target environment with object code generated by the operating system assembler, etc. In all cases, the linker may be run only once per executable image. The input to the linker must contain exactly one main program but may contain many object files derived from units.

F.1.4 Libraries

Object files in ".obj" format may or may not be libraries. The result of a run of the code generator is an ".obj" which is not a library, although it is possible that such a file contains object code with corresponds to many subroutines. The main difference between ".obj" files which are libraries and those which are not libraries is that the linker includes all of the object code from non-library input files but only that object code which is referenced from library input files. The determination of what is referenced is made based on unresolved external code references in previous input files to the linker. Therefore the order that files are presented to the linker is important.

When linking **Pascal** programs, the run time library provided with the system, **paslib.obj**, must be the last input file to the linker.

F.1.5 Error Messages

The **Pascal** system contains a file of compile time error messages. If this file is given the appropriate name, the compiler will generate English error messages along with error numbers. If not, the compiler will only give error numbers. The name of this file differs from one implementation to another

and can be found by referring to the following appendix.

F.2 Command Line Directives and Compiler Options

The **Pascal** compiler front end is invoked to compile a source file named "prog.pas" (other file name endings required on other systems) with a command line of the form:

pascal prog.pas { options... }

Any number of command line options may appear and they may appear in any order. The possible command line options are:

+q −q — Show more (−q) or less (+q) information on the progress of the compile to the user. The default setting varies among different implementations.

+p −p — Prompt (+p) or don't prompt (−p) to the standard input in the case of a compile time error. The default setting varies among different implementations. Prompting mode is useful so that error messages do not fly off CRT screens but is awkward when compiling in background mode.

+f −f — Generate code for the Sky floating point hardware board (+f) or generate code for software floating point (−f). This option is only enabled in systems which support the Sky board and will result in an error if not enabled. The default is −f, no floating point hardware. Note: If the Sky floating point hardware interface is to be used, *the entire* program must be compiled with the +f flag set and the resulting object code must be linked with **sky.paslib.obj** instead of **paslib.obj.**

−lfname — Create a listing file of the source program in the file named fname.

−efname — Place a summary of the compile time errors on file named fname.

−ifname — Name the ".i" file fname. If this option is not provided, the ".i" file when compiling a source program named prog.pas is named prog.i.

Under certain operating systems the code generator is directly invoked by the **Pascal** compiler front end. In this case, there is an additional command line option.

−ofname — Name the ".obj" file fname. If this option is not provided, the ".obj" file when compiling a source program named prog.pas is named prog.obj.

Under systems in which the code generator is not directly invoked by the **Pascal** compiler front end, the code generator is invoked using a command of the form:

code prog.i { optionalfname }

where leaving off the optional file name results in an output file named prog.obj. If the optional file name is provided, the output file is named optionalfname.

See the appendix which follows for a description of command line arguments and options related to the linker.

F.3 Linking Programs which Utilize C and FORTRAN

There are certain rules which must be observed by programmers wishing to combine object code compiled under more than one language processor. Throughout the following discussion, **Pascal**, **FORTRAN**, and **C** refer to the *SVS implementations* of these languages.

F.3.1 What Language must Supply the Main Program

In all cases in which **FORTRAN** code is present, the main program must be **FORTRAN**. In the case where **Pascal** and **C** are to be present, either language may supply the main program. If the C system is not **SVS C**, then the main program must be **Pascal**.

F.3.2 Referring to the Command Line Arguments

In all cases in which the command line arguments are to be referenced from **C**, **C** must provide the main program. This is a consequence of the fact that command line arguments are "parameters" to the **C** main routine. Command line arguments are available from **Pascal** and **FORTRAN** regardless of which language provides the main program.

F.3.3 Dynamic Memory Allocation and Deallocation

A program may utilize the **C** library memory allocation and deallocation package (malloc, free, etc.) providing that **Pascal** components of the program do not call **release**. Similarly, **Pascal** components should not call **release** if **FORTRAN** components performing any I/O are present. If the C system is not **SVS C**, then the C routines *must not* utilize any dynamic memory allocation or deallocation directly or through the operating system run time library.

F.3.4 Parameter Conventions

The calling convention in **C** is such that parameters are pushed in "reverse" order from the order in which they appear and the *calling* routine is responsible for popping parameters off the stack after the call returns. **Pascal** and **FORTRAN** push parameters in order and the exit code of the *called* routines is responsible for popping off its parameters. **Pascal** contains a "cexternal" declaration (similar to **Pascal** "external") which generates calls to **C** routines in which the parameters are popped off at the calling site after the subroutine returns. The parameters must appear in reverse order in the **Pascal** call as compared to the order expected by **C**. There is no direct language support for calling **C** from **FORTRAN** or **Pascal** and **FORTRAN** from **C**, but parameterless routines or assembly language interfacing routines can be utilized for these purposes. It is often easiest to go through **Pascal** when calling **C** from **FORTRAN**.

F.3.4.1 Calling C from Pascal

The **Pascal** program should contain a **cexternal** declaration with all parameters four bytes in length (except floating point which should be double precision). Addresses may be passed by specifying the parameters to be var parameters. The following declaration in Pascal

```
function cfunct(i,j: longint; d: double): longint; cexternal;
```

can be used to call the **C** function

```
cfunct(d,j,i)
int    i,j;
double d;
{
if (d == 0.0) return(i+j); else return(i-j);
}
```

No assembly language is necessary to link these routines. Note: on some operating systems the **C** system prepends underscores to external names and the **Pascal** declaration would have to be for a function named _cfunct rather than a function named cfunct.

F.3.4.2 Calling Pascal from C

There is no way to tell the **C** system that an external reference is to a non **C** routine. Therefore, using the types of the variables from the previous example, a **C** call of the form

```
i = pasfunc(d,j,i);
```

would require an assembly language "wrapper" of the form

```
        .text
        .globl  pasfunc
        .globl  PASFUNC
pasfunc:
        movl    sp@+,savera
        jsr     PASFUNC
        subl    #16,sp
        movl    savera,-sp@
        rts
        .bss
savera: . = . + 4
```

to call a **Pascal** function declared with the header

function pasfunct(i,j: longint; d: double): longint;

The important items to note are: **Pascal** entry point is in upper case, **C** external reference is in the same case as the programmer specified. The .globl for the **C** entry point may need a prepended underscore on some operating systems. The "wrapper" will *not* work if the interlanguage call is recursive. The **C** calling site expects to pop off 16 bytes of parameters after the call returns, but the **Pascal** function has already popped off the parameters. Therefore, the "wrapper" decrements the stack pointer by the amount the calling site expects to pop off.

The exact syntax of the assembly language will vary from system to system. In general the object code for "wrapper"s is linked into the executable program at the last linking step of the compile. Normally, a wrapper is required for each **C** to **Pascal** call.

The above procedure will not work with C systems other than SVS C because other C systems expect called subroutines to preserve different registers than **Pascal** functions preserve. In this case, the "wrapper" must be enhanced to preserve the registers required by the calling C language subroutine.

F.3.4.3 Calling FORTRAN from Pascal

It is straight forward to call **FORTRAN** subroutines from **Pascal**. The called routines should be declared to be **external** in the **Pascal** compilation with formal parameter declarations which match **FORTRAN** parameter conventions. In particular, **Pascal var** parameters will match the **FORTRAN** call by reference convention. If the receiving **FORTRAN** routine expects a character parameter, it will be necessary to pass the length of the **packed array of char** as an explicit two byte value parameter (as described in the parameter passing section of the **FORTRAN** reference manual). Note: **Pascal** strings are not compatible with the **FORTRAN** character datatype.

F.3.4.4 Calling Pascal from FORTRAN

When calling an external routine from **FORTRAN**, it is merely invoked without any special declaration. This called routine may have been written in **Pascal**. In the event that it is, the routine should be written with formal parameters declared in the manner which is consistent with what **FORTRAN** would expect from a receiving routine written in **FORTRAN**. Pascal formal parameter declarations are adaquate for expressing all of the interfaces expected by **FORTRAN** calling sites.

F.3.5 Run Time Libraries

When linking multiple languages, the last input file provided to the linker must always be paslib.obj. Immediately preceding paslib.obj must be clib.obj and ftnlib.obj, in either order. The former must be present if **C** is present and the latter must be supplied if **FORTRAN** is contained in the program being linked.

F.3.6 Upper and Lower Case External Naming Conventions

It is the convention in **Pascal** and **FORTRAN** to upper case all external names *except* routine names which are declared **cexternal** in **Pascal**. These names are passed directly to the linker as they appeared in the **cexternal** declaration. In **C**, upper and lower case letters are distinct, so it is the convention to pass letters directly through as they were supplied by the programmer. For interfacing purposes, use upper case names in **C**, or use cexternal in **Pascal**, or use assembly language to bridge the naming conventions.

F.3.7 Prepended Underscore to External Names

Some of the operating system environments prepend the underscore character to **C** external names. **Pascal** cexternal names do not get underscore prepended to them in any environment, but **Pascal** accepts underscore as a letter so that the user may generate **Pascal** cexternals with leading underscores.

Appendix G – UNIX Operating System Specific Information

Although the **SVS Pascal** system appears to be almost identical under a wide variety of operating systems, there are minor differences, particularly related to the linker and in operating procedures, among the various environments. This appendix will provide the implementation dependent details related to **SVS Pascal** running under the UNIX operating system.

G.1 Compiling a Simple Program

The instructions provided here for compiling and linking a **Pascal** program reflect the system as it is released by SVS. Some vendors of the system provide additional utilities for sequencing compiles for which there may be separate documentation.

Appendix F of this manual described in some detail the components of the **SVS Pascal** system. For most **Pascal** programs, the following simple procedure will be completely adequate for sequencing a compile:

Create a "shellscript" called Pascal with the following commands:

```
set -e
pascal $1.pas
code $1.i
ulinker -l $1.o $1.obj paslib.obj
cc $1.o
mv a.out $1
rm $1.o $1.obj
```

To compile a **Pascal** program in a file named prog.pas, execute:

```
Pascal prog
```

The **Pascal** program and the shellscript can be created using the system text editor. The "mode" of the shell script should include execute permission (i.e. chmod +rwx Pascal). The shellscript assumes that pascal (the **Pascal** compiler front end), code (the code generator), and ulinker (the linker) reside in the system in directories from which they can be executed. The shellscript also assumes that paslib.obj is the correct pathname for accessing this file. These names will most likely have to be changed to reflect the location of these files

on your system.

The lines of the shellscript do the following: The set −e causes the compiling sequence to terminate after an error is detected. The next lines run the front end and code generator on files whose names are derived from the command line in which the shellscript is invoked. The linker is run (in its simplest form, see below for more details) with −l inhibiting a linkmap listing file, with output file $1.o, and with two input files, including the SVS supplied library. Ulinker produces a file which is then linked to those UNIX system calls which are utilized by the program in the cc step (which invokes the UNIX system linker). The final two lines rename the executable program and remove the unlinked object code files.

G.2 Error Message File

SVS Pascal includes a file called pascterrs which should be placed in either the /lib or /usr/lib directory. This will allow the compiler to display English messages for errors which it detects.

G.3 Ulinker

Under UNIX, ulinker is the SVS linker. The general operation of the linker is described in Appendix F. This section will describe in detail the modes of operation of ulinker and its load map listing option.

G.3.1 Ulinker Inputs

Ulinker links object code in ".obj" format, including libraries. In addition, ulinker accepts input from the command line or interactively as described below.

G.3.2 Ulinker Outputs

Ulinker creates partially linked object code in UNIX ".o" format as its primary output. Optionally, ulinker can produce a listing file which is a load map of global entry points in the created ".o" file. The form of this map and information contained in it is best described by the following example with subsequent explanations:

Example of Ulinker Listing File

```
Linking segment '          ' (670)
MC68000 Unix Object Code Formatter   22-Aug-83

File: prog.o

Memory map for segment '          '

COMPUTES  -  COMPUTES    00001E
FAIRLYSI  -  FAIRLYSI    000054
$START    -  $START      000054
%P830701  -  %P830701    000082
%_TERM    -  %_TERM      0001E0
%_END     -  %_END       0001E2
%_VERS    -  %_VERS      0001E6
%I_MUL4   -  %I_MUL4     0001EC
%I_MOD4   -  %I_MOD4     00021C
%I_DIV4   -  %I_DIV4     000228

No:  Segment:    Size:
 0.  '          '  00029E

Start Loc  =  000054
Code Size  =  00029E
Global Size  =  000006
```

Explanation of Ulinker Listing File

The listing file was generated from the following **Pascal** program:

```
program fairlysimple;
var i: integer; li: longint;

   procedure computesome;
   begin
   li := (li * li) mod 99999;
   li := li div 17;
   end;

begin
li := 2;
for i := 1 to 100 do
   computesome;
end.
```

The segment named by 8 blanks had 670 (decimal) bytes in it. Under UNIX there is no reason for programmers to explicitly deal with segments, since ulinker handles this automatically.

There were ten entry points in the linked files. Eight of these were pulled out of the library and two are recognizable as user function names. The addresses of these entry points are given in hex and are text area relative, *but will be further relocated by the cc step of the compilation.* The relative addresses (distance between them) will remain intact through the cc step.

There would be a data areas shown associated with each of the **units** in the link, mapped to the data or bss area depending upon whether the area is initialized at compile time (which is possible using **FORTRAN** block data and named common). Sizes and locations of these data area listings are in hex and relative to the start of the data or bss area as appropriate.

G.3.3 Running Ulinker from the Command Line

The command line form of running ulinker is:

ulinker listfname outputfname inputfname { inputfname ... }

where the optional listing file is created on a file named listfname providing that listfname is not equal to -l (no listing file to be created directive). The command line arguments are positional. No file name suffixes are enforced by ulinker in this mode so complete file names must be entered.

G.3.4 Running Ulinker Interactively

It is often not convenient or not possible to have a command line which is long enough to have all of the input files listed. In this event, ulinker can be run interactive. Execute ulinker without any command line arguments and it prompts:

Listing file -

Any file name provided creates the listing file. Enter just return to suppress the optional listing file. The next prompt is:

Output file [.o] –

Ulinker requires an output file. If the file name provided does not end in ".o", ulinker will append this file name extension onto the name which is input. Following this prompt, ulinker will repeatedly prompt:

Input file [.obj] –

for its input files, until a plain return is typed, indicating that the input file list is completed. Ulinker will append the ".obj" suffix onto input file names if it is not supplied by the user. Running in this mode, there is no limit on the number of input files which ulinker can process.

G.3.5 Running Ulinker with Standard Input Redirected

With many input files, it is most convenient to operate ulinker in its interactive mode with standard input redirected. For example, run ulinker as follows:

```
ulinker <cmd
```

where the file cmd contains a line for the listing file name, a line for the output file name, lines for the input file names, and a blank line to terminate the input file list.

G.3.5.1 Symbol Table Information Placed in Output File

Utilizing the UNIX utility nm it is possible to examine the symbol table information placed in the output file by ulinker. In general, all entry points which are not local to another procedure (a situation which only occurs in Pascal) are placed into the ".o" file symbol table. All entry points appear in the ulinker listing file, including those which are Pascal local procedures. There are also symbol table entries for unresolved external references and for the program entry point (named _main under UNIX).

G.3.6 Treatment of Unresolved External References

Unresolved external references are passed through into the output file for potential linking in the cc step of the compile. In the event that these references are not resolved at that stage, an error message is generated then.

G.3.7 Segments

Under some operating systems other than UNIX, the **SVS Pascal** system contains a meaningful object code concept referred to as segments. Under UNIX, there are segments in the object code, but they are not semantically meaningful. Ulinker automatically creates segments as needed and there is no reason for the user to do anything explicitly about creating and/or naming segments.

G.3.8 Errors Detected by Ulinker

Most of the error messages which come out of ulinker are completely self explanatory. The error message:

```
*** In data area named ABC
*** at offset 999 bytes into that data area
*** Fatal Error - overlapping data initialization
```

is caused by user programs initializing the same location in the named data

area more than once. The error message:

*** Error – Double defined: ABC

is caused by the same entry point name being used in more than one input file. Only 8 characters are significant for the linker. The error message:

*** Error – Double defined unit

is caused by linking more than one unit with the same name. The link name for Pascal units begins and ends in slashes and contains the six initial characters of the Pascal unit user name between the slashes. This facilitates initializing Pascal unit globals using FORTRAN named common and data statements. One consequence of this link naming convention is that only six characters of the user unit name are utilized for resolving naming conflicts. The error message:

*** Error – Multiple start locations

is caused by having more than one main program among the input files.

G.4 Linking to UNIX Assembly Code

It is normal for the output of ulinker to contain unresolved external references to UNIX system calls (such as _open, _close, and _write). These are resolved by the cc linking step by using the operating system default library of UNIX object code. The user may do the same kind of linking to UNIX assembly code by providing the assembly language source as an additional argument to the cc compilation step which will automatically invoke the operating system assembler.

One limitation on code which is linked in with code generated by the SVS languages is that no UNIX system calls on malloc, free, sbrk, or related routines (directly, or through other linked in routines) may be used. The SVS languages handle the UNIX break area of memory, including versions of malloc and free in the SVS C library, in a manner which is not fully compatible with the UNIX routines.

User's should also beware of differing floating point formats. Some of the UNIX systems do not use IEEE format floating point. In this event, passing floating point values will result in strange results.

It is not guaranteed that I/O will work as expected across language boundaries, particularly with respect to object code generated by non SVS systems.

Any code linked into programs generated by the SVS languages must obey the register and calling conventions assumed by the system. In particular, all called routines must preserve registers D3 through D7 and A2 through A6. More details on the calling conventions are provided in the appendix on data representations.

G.5 Argc and Argv

Under UNIX, the name of the program is the first argument in the argv list of the invoked program, that is argv[1]^. Argc is always at least 1. The first user supplied command line argument is argv[2]^. This is sometimes confusing for UNIX programmers who are more used to seeing the name of the invoked program as the zero'th argv in the **C** programming language and the first user supplied command line argument as the one referenced using array index 1 on the argv array. The **Pascal** numbering scheme is consistent with the fact that argv is a one origin indexed array.

G.6 Features not Implemented Under UNIX

The following features of **SVS Pascal** are not implemented under UNIX: call, unit I/O (such as unitread, unitwrite, etc.), and memavail (not implemented under most UNIX implementations).

G.7 Return Values from Pascal Programs

A **Pascal** program can issue the call:

halt(integervalue)

to generate a UNIX system termination code equal to the value specified. If a zero value is provided, UNIX will consider that the program "succeeded", otherwise UNIX will treat the process as having terminated with an error. This is useful for interacting with shellscripts which test the UNIX error flag after executing programs written in Pascal.