A SWL APOLOGIA

J. Keffe
L. Kerr
J. Merner
C. Schwarcz

15 September 1973

TABLE OF CONTENTS

TABLE OF CONTENTS

1.0 INTRODUCTION

The Software Writers' Language (SWL) Project was originally charged with the responsibility of designing a systems implementation language which would be as compatible as possible with Pascal. This proved to be a somewhat ambiguous goal, as there are at least six sources which could reasonably be used as a definition of the Pascal language, and no two of them agree in all respects.

The Pascal references which were used during the design of the SWL are:

1. N. Wirth, "The Programming Language Pascal", Acta Informatica 1, 35-63 (1971). This paper describes the original Pascal language.
2. N. Wirth, The Programming Language Pascal (Revised Report), E.T.H. Zurich (November 1972). This report describes a language which is basically similar to that described by (1), but which differs substantially in certain areas.
3. C.A.R. Hoare and N. Wirth, An Axiomatic Definition of the Programming Language Pascal, E.T.H. Zurich (November 1972). This report represents an attempt at a rigorous, formal definition of the language described by (2).
4. The June 1972 Pascal 6000 Compiler. This is a compiler which implements the language described by (1).
5. The December 1972 Pascal 6000 Compiler. This compiler contains some of the modifications required for the language described by (2). It was intended to bridge the gap between (1) and (2) until a new compiler was completed for (2).

6. The March 1973 Pascal Stack Machine Compiler. This compiler implements most of the language described by (2), and compiles code for a hypothetical stack machine. The compiler will ultimately evolve into a Pascal 6000 compiler for (2).

In the following sections of this document, an attempt is made to list the changes that were made to Pascal in the design of the SWL, and the reasons for these changes are discussed. The changes are described primarily relative to the revised Pascal report and the December 1972 Pascal 6000 compiler, as these are the reference sources with which CDC and NCR Pascal users are most familiar.

The changes are grouped into five sections: Control Structures, Data Structures, Syntactic Changes, Machine Dependent Features, and Miscellaneous Changes. These categories are rather arbitrary, and were chosen merely for ease of presentation. Some overlap exists between different groups: procedure variables, for example, represent an enhancement to the control structures as well as being a new data type. In such cases, the changes were placed arbitrarily in one group or the other.

Wherever possible, reference is made to the appropriate sections of the SWL Specification document which describes the language features in detail. Such references take the form of a section number enclosed in brackets (e.g., [5.3]).

2.0 CONTROL STRUCTURES

The changes discussed in this section include new control statements (e.g., loop, return, resume, etc.), new data types that are related to control mechanisms (e.g., label, procedure and coprocess variables), and modifications of existing Pascal control structures.

2.1 LOOP AND ESCAPE STATEMENTS [10.2.4, 10.3.5-10.3.7]

The loop statement and the escape statements (continue, exit, and return) were introduced to replace the use of the goto statement in common programming situations which do not require the full generality of the goto. This will encourage the writing of more highly structured programs and should improve both readability and reliability.

The loop statement has the effect of repeatedly executing the statements contained within the loop-loopend pair. This action continues until altered by the execution of some control statement within the loop – typically, an exit statement which causes control to pass to the statement following the end of the loop.

The continue statement is used to start a new iteration of a for, repeat, while, or loop statement. The exit statement is used to terminate the execution of a structured statement, and may also be used to cause control to exit from an enclosing procedure. The return statement is provided as a convenient form of exit from the nearest enclosing procedure.

While the loop and exit statements are not contained in any Pascal language descriptions, they are implemented in the May 1973 Pascal stack machine compiler.

2.2 INCREMENT ON 'FOR' STATEMENT [10.2.7]

In order to allow greater flexibility in the control of the index variable, an optional by clause was added to the for statement. When specified, this clause allows increments of other than +1 and -1.

2.3 PROCEDURE VARIABLES [4.1.9, 10.3.1]

While Pascal allows only formal parameters to be declared with type procedure, in SWL this concept has been extended so that any variable can be declared to be a procedure variable. This extension was required to allow the implementation of loaders, debug packages, and table-driven systems.

2.4 LABEL VARIABLES [4.1.8, 10.3.8]

Label variables are provided primarily to allow satisfactory handling of error conditions. Label variables allow dynamic stack unstacking, including the situation in which control is returned to a procedure in a different compilation unit, and also provide for alternate return points from procedures when they are used as formal parameters.

2.5 'ORIF' CLAUSE ON 'IF' STATEMENT [10.2.3]

A programming construct that arises frequently in Pascal is the following:

```
if COND1 then STAT1
else if COND2 then STAT2
    :
    :
else if CONDN then STATN
```

The problem with this construct is that in reality the else clauses become more and more deeply nested, and this artificial nesting has two unfortunate implications:

1. An automatic source code formatter will make the nesting explicit by progressively indenting each alternative across the page. This is especially serious if there are more than just a few alternatives.
2. The unique ending delimiter principle adopted for SWL (described later in Section 4.2) forces the programmer to code numerous ifend symbols after the last alternative.

These difficulties are avoided by using the orif construct, which is essentially equivalent to else if without its inherent nesting.

2.6 'ELSE' CLAUSE ON 'CASE' STATEMENT [10.2.8]

It often occurs that only a few cases in the possible range of a case variable require special handling, while all the others may be grouped together and handled in a uniform way (frequently as an error condition). The else clause provides a way of describing this situation concisely and conveniently.

2.7 FUNCTIONS DECLARED AS PROCEDURES [4.1.9, 8.0, 8.3]

Although the Pascal concept of not allowing functions to cause side-effects has some merit, it was decided that such a restriction in SWL would be too severe. Functions in SWL can therefore be viewed as procedures which return a value, and hence they can call other procedures and alter nonlocal variables.

In order to make the distinction between SWL functions and Pascal functions explicit, all functions in SWL are declared as proc's which return a value.

2.8 COPROCESSES [4.1.10, 8.0, 10.3.2-10.3.4]

A set of coroutines is a set of concurrently active procedures which link together through a mechanism which combines the features of "call" and "return". Some programs can be coded conveniently only through the use of coroutines - an example is the case of a compiler that performs macro expansion and syntactic analysis in the same pass. To simulate coroutines in a block structured language is difficult and requires dropping into implementation dependent code.

Coprocesses are a generalization of coroutines in which more than one instance of a particular coroutine may be active at a time. This provides the equivalent of synchronous multiprocessing, in which a single thread of control is switched from one process to another explicitly by the programmer. The coprocess facilities therefore combine the advantages of coroutines and parallel processes, while avoiding the additional overhead required by asynchronous multiprocessing.

Provision is made in SWL for creating a new coprocess and storing its identity in a coprocess variable, transferring control to a coprocess and saving the state of the current one, and destroying a coprocess.

3.0 DATA STRUCTURES

The topics discussed in this section include data types, data structuring methods, attributes of variables, and storage allocation control.

3.1 VARIABLE BOUND ARRAYS [4.2.2]

Pascal requires the bounds of all arrays to be known at compile time, making it impossible to code procedures which accept array parameters of arbitrary size or to create array variables whose bounds are not known until run time. This restriction has been relaxed in SWL with the introduction of variable bound arrays and adaptable arrays.

A variable bound array is a local array variable, one or more bounds of which is an expression which can only be evaluated at run time. In this case, the array bounds are evaluated upon entering the block in which the array is declared.

Adaptable arrays contain one or more indefinite bounds, indicated by coding an asterisk instead of an actual bound. They may be used as formal parameters, in which case the indefinite bounds assume the bounds of the actual parameter, as arrays which are to be explicitly allocated, in which case the actual bounds are specified in the allocate statement, or they may be made to designate an actual array by means of the designational assignment statement.

3.2 SLICE NOTATION [10.1.1]

Since Pascal treats character strings as arrays of characters, the most convenient way of denoting a substring is through the use of array slice notation. An array slice is an array reference in which a subscript range is specified in place of the usual index expression. The resulting subarray can then be used in an array assignment statement,

passed as a parameter to a procedure, or used generally wherever an ordinary array can be used.

3.3 STORAGE CLASSES 5.2.1, 7.1.1.2, 7.1.1.3

Since SWL is required to support separate compilation of procedures, it was necessary to provide a number of new storage classes. These storage classes are:

1. static - The storage for the variable is allocated at load time, and the value of the variable is maintained from block exit to block reentry.
2. xdcl - The variable is treated as a static variable, and in addition its name is known externally and may be accessed from other compilation units by variables declared as xref.
3. xref - The identifier being declared refers to an xdcl variable declared in another compilation unit.
4. external - Storage for an external variable is shared in common with external variables declared with the same name in other compilation units.

3.4 SEGMENTS [7.1.1.2, 7.2]

To allow greater control over the allocation of static variables and permit improved locality of memory references, provision is made in SWL for naming memory segments and specifying that certain variables and procedures are to be stored in particular segments. In addition, a segment may be restricted to some combination of read, write, and execute access privileges, in which case the compiler will assist in detecting violations of these privileges.

3.5 READ-ONLY VARIABLES 7.1.1.1

A need exists for variables whose value is determined when the variable is declared, and is then to be left unaltered. Examples of this are the tables for a table-driven compiler, an array of character strings for error messages, etc. The advantage of being able to specify that a variable is not to be altered subsequent to initialization is not only that the compiler can assist in detecting violations of this intent, but also that more extensive optimization can be performed. This should result in improved efficiency as well as reliability.

3.6 INITIALIZATION [7.1.2]

Although the Pascal language has no provision for initializing variables other than by explicit assignment statements, the Pascal 6000 compilers provide the value statement which can be used for this purpose. This method was discarded in SWL in favor of an initialization clause as part of the variable declaration statement. In SWL, therefore, the initialization information is specified in the same place as other declarative information for a variable, greatly improving the readability of the program.

3.7 STRUCTURED DATA CONSTRUCTORS [6.2]

A structured data constructor is a mechanism for describing some value of a structured type. Pascal already has such a mechanism for sets; in SWL the concept was extended to include arrays and records as well. The prime justification for structured data constructors is their use for initializing structured variables, but they can be used to advantage in expressions as well.

Essentially three kinds of data constructors are provided: one that builds an ordered list of elements (i.e., an array or record), a second that builds an unordered set of elements (i.e., a set), and a third that builds a value of any specific structured type.

It was difficult to decide on suitable bracketing characters for the first two constructs. Braces were chosen for set constructors as being the most natural with respect to traditional mathematical notation. Although either parentheses or angle brackets would have been suitable for array and record constructors, both lead to ambiguities in the language so square brackets were chosen instead.

3.8 UNION OF TYPE [4.2.4]

A major decision in the design of SWL was to discard the Pascal concept of variant records, and to replace it with the concept of union of type as used in Algol 68. The reasons for making this change are:

1. Whereas fields of a Pascal variant record can be accessed under the guise of any of the variants without any checking of the tag field being imposed by the compiler, a SWL union of type can be accessed only after first ensuring that the actual type of the value is the one intended. In this respect the SWL construct is more restrictive than the Pascal construct, and it should eliminate some programming errors as well as some questionable programming practices.
2. Union of type allows procedures to be written which accept parameters which can assume values of more than one type.

3. Different variants of a variant record frequently have several fields in common. Unless they appear in the fixed part of the record, Pascal requires that each of these fields have a unique field identifier. This restriction does not exist with union of type.

3.9 HEAPS, STACKS, QUEUES, AND SEQUENCES [4.2.5-4.2.8, 10.4]

Heaps, stacks, queues, and sequences were introduced as new data types in SWL to provide more control over storage allocation (improving memory reference locality) and to facilitate the handling of certain data structures.

Heaps are areas of memory out of which variables can be allocated and freed. Since heaps can be local to a block, any unfreed storage in the heap is automatically returned when control exits from the block.

Stacks, queues, and sequences are also areas of memory in which variables can be stored and removed, but they assume an additional structuring of these variables. Stacks and queues are intended for the commonly used structures that their names imply, while sequences are used to store any data structure which is sequentially decodable by an algorithm provided by the programmer. Sequences are useful for describing structures such as blocks of variable length arrays, where each array is preceded by an integer indicating its length.

3.10 RELATIVE POINTERS [4.1.7]

Relative pointers were introduced to allow data structures to be moved from one area of memory to another without requiring all pointers to the data to be updated. An additional advantage of relative pointers is that they can be more compact to store than regular pointers.

Each storage reference using a relative pointer must specify which memory area the pointer is to be based on for that reference.

4.0 SYNTACTIC CHANGES

The changes discussed in this section are those that involve a change in syntax only, and do not have any effect on the functional capabilities of the language.

4.1 CHARACTER SET

It was assumed that ASCII will be the official character set for the Integrated Product Line. Since Pascal uses several characters that are not contained in ASCII, some character set changes were required. These changes are summarized in the following table.

Pascal symbol	SWL symbol
∨ ∧ ¬	| & ~
≤ ≥ ≠	<= >= /=
↑	^

Since some SWL symbols are outside the CDC 63-character ASCII subset, alternate representations are provided to allow the compiler to be accommodated on CYBER 70 equipment.

SWL symbol	Alternate representation
| & ~	OR AND NOT
{ }	[: :]

4.2 UNIQUE DELIMITERS [10.2]

One of the changes that gives the greatest appearance of incompatibility with Pascal is the concept of unique ending delimiters. As a result of adopting this policy, each structured statement is terminated with a symbol that is unique to that statement type. For example, a while loop in Pascal such as

```
while   COND   do
    begin
        S1;
        ...
        SN
    end
```

in SWL becomes

```
while   COND   do
    S1;
    ...
    SN
whilend
```

The reasons for adopting the principle of unique ending delimiters are:

1. The added redundancy in the source text allows the compiler to detect unbalanced delimiters at a much earlier stage, permitting it to issue more meaningful diagnostics and perform more intelligent error recovery.

2. The unique delimiters are a great aid to the human reader in matching the beginning and end of structured statements. This is especially true if the program is also formatted into paragraphs that reflect the nesting structure.

3. Structured statements such as while and for are usually composed of a list of component statements, rather than just a single statement. In Pascal the list of statements must be explicitly bracketed by a begin-end pair. In SWL this bracketing is inherent in the structured statement itself, which in most cases provides for a more natural and readable construct.

It should be noted that it is a straightforward task to mechanically translate an existing Pascal program into one which uses unique delimiters.

4.3 DECLARATIONS [5.1, 6.1, 6.3.1, 7.1, 7.2, 8.0]

Pascal requires a strict ordering of constant definitions, type definitions, variable declarations, and procedure declarations. It was decided that this ordering was not acceptable for SWL, because of the requirement to be able to group together all the constant, type, variable, and procedure declarations for a particular functional module. This is necessary so that the declarations can be stored away as standard source text, and included into user programs as required.

As well as allowing arbitrary ordering of the different kinds of declarations, it was decided that each declaration section should be made into a single statement, with the component declarations being separated by commas instead of semicolons as in Pascal. This guards against the possibility of a minor error causing the misinterpretation of an entire declaration section, and also seems to be a more consistent use of the semicolon.

4.4 IDENTIFIERS [3.1]

To improve readability, the maximum length of SWL identifiers was increased to 31 characters, and the underscore was introduced as an alphabetic character. The pound sign, dollar sign, and at sign (#, $, and @) are considered to be alphabetic and may also be used within identifiers.

4.5 LABEL IDENTIFIERS [10.0]

Labels in SWL are denoted by identifiers, rather than by integers as in Pascal. Identifiers can be made far more meaningful, and their use should lead to more readable programs.

4.6 INTEGER CONSTANTS [6.3]

SWL provides for describing integer constants in decimal notation or in base 2, 4, 8, or 16 notation. The additional notations are required for describing machine dependent values in a convenient and readable fashion.

4.7 COMMENTS [3.1]

The comment delimiters were changed from braces in Pascal to quotation marks (") in SWL. This decision was arrived at for the following reasons:

1. Bracketing symbols were in high demand for use as structured data constructors, and braces seemed the most appropriate for set constructors.

2. Braces do not appear in the CDC 63-character ASCII subset. Although an alternate representation could have been provided for use with CYBER 70 equipment, it was felt that for such a common language element as comments a single, uniform representation was preferable.

3. The quotation mark was not required for use as any other symbol in the language.

4. Quotation marks proved to give adequate visibility to comments, and were found to have a natural and pleasing appearance.

5. PL/I-style comments (/* and */), the only other serious contender, were rejected as they were felt to be unnatural and ugly.

Another change in the syntax of comments was motivated by a concern for the chaotic effects of inadvertantly dropping the closing delimiter of a comment. (Note that a similar problem exists even when the opening and closing delimiters are distinct. This case is actually more insidious, since the compiler recovers at the end of the next comment and the error can go undetected.) In order to detect this situation at an early stage, the restriction was made that semicolons may not appear within comments.

5.0 MACHINE DEPENDENT FEATURES

The language features discussed in this section are those that are concerned with the hardware representation of variables or with actual machine instructions. This is not meant to imply, however, that programs which use any of these features are necessarily machine dependent. It is possible, for example, to write a machine independent memory allocation procedure using type cell and the size, alignment, and location functions. In fact, crammed records were introduced for the very purpose of allowing convenient transfer of information between machines of different architecture.

5.1 'LOC' FUNCTION [11.2.11, 11.3.1, 13.2]

It is sometimes necessary to be able to reinterpret some arbitrary part of memory according to a particular type. This facility is provided in SWL by the loc function, which returns the location of a variable. Use of the loc function is restricted to a direct assignment to a pointer variable, which may then be used to access memory according to the type of the pointer.

The loc function may be applied only to directly addressable variables, thereby excluding elements of packed structures.

5.2 CELL TYPE [13.1.1, 11.3.2-11.3.4]

It was felt that there was a need for untyped storage in SWL, to be used for such applications as memory allocation routines, storage areas for interpreters, etc. A cell is defined to be the smallest unit of memory directly addressable by a pointer, and could be a word, a byte, or a bit, depending on the architecture of the particular machine.

The only operation defined on variables of type cell is that of assignment. Other types of access to cell variables can be made in conjunction with the loc function.

In order to allow cell variables to be used in a machine independent way, a number of standard procedures are provided which relate the size of a cell in a particular implementation to the other SWL data types. These functions are:

1. size (arg) - returns the number of cells required to contain a variable of the same type as 'arg'.
2. maligned (arg, offset, base) - provides the offset and base alignment of 'arg' in terms of cells.
3. malignment (arg, offset, base) - provides the offset and base alignment required for a variable of the same type as 'arg'.

5.3 CRAMMED RECORDS [11.3.2, 11.3.4, 13.1.2]

Crammed records are provided to accommodate those situations where the programmer must have bit by bit control over the representation used for a data structure. For the representation of a crammed record, the bits required to store a particular field follow immediately after the previous field, regardless of any natural storage unit boundaries.

Explicit control over the size of a field and its alignment is achieved through the optional use of the width and maligned attributes.

5.4 MACHINE CODE [14.4]

Although the need to escape to machine code should be comparatively rare, it was still felt necessary to allow the generation of machine code instructions directly in SWL. In order to control the use of this facility, all machine code statements must be contained within 'code blocks', which are groups of SWL statements and machine code statements bracketed by code-codend symbols. The machine code statements are distinguished by prefixing them with an exclamation point, and their syntax and semantics are left to the code generator for the particular machine involved.

5.5 MACHINE DEPENDENT TYPES [14.1, 14.2]

Apart from the requirement to be able to generate specific machine code instructions, there is the more common need to deal with machine dependent types and storage classes. Examples of these would be a page table entry type, register storage class, etc. All usage of such machine dependent types and storage classes must appear within 'code blocks'. Variables declared with machine dependent storage class but normal type may be used in normal SWL statements, but variables declared with machine dependent type are restricted to assignments, equality tests, and machine code statements.

6.0 MISCELLANEOUS CHANGES

6.1 COMPILE-TIME EXPRESSIONS

SWL allows expressions that can be evaluated at compile time to be used wherever constants are allowed. This reduces the amount of hand computation required of the programmer and provides constants with a greater information value (e.g., 4* page_size instead of 4096).

6.2 'BEGIN' BLOCKS [10.2.1]

It was decided to provide begin-end blocks which allow local declarations. This permits more efficient use of storage, and allows macros to be written with variables which are local to the macro body.

6.3 OPERATORS [9.2]

The following changes were made to the operators:

1. In Pascal the division operator (/), when operating on two integers, results in a quotient of type real. Since it was felt that real numbers have minimal utility in a systems implementation language, this behavior was judged to be undesirable, and that the result should be that of integer division with truncation. This definition was adopted, and the Pascal integer division operator (div) was removed from the language.

2. It was decided that the 'not' operator should be extended to sets in order to be able to determine the complement of a set. The tilde (~) is used to represent this operator.

3. It was felt that the minus sign, being an arithmetic operator, was not appropriate for set difference. The tilde (~), already a unary set operator, was extended to serve as the set difference operator.

4. An 'exclusive or' operator (xor) was added to the language.

5. A rep operator is provided in SWL to form a list of repeated values, e.g.,

   EXP1 rep EXP2

   is equivalent to

   EXP2, EXP2, . . ., EXP2

   where EXP2 is repeated EXP1 times.

6.4 PARAMETERS [4.1.9, 8.2, 10.3.1]

While the default method of passing parameters in Pascal is by value, it was felt that the more efficient method of passing parameters by reference (used for var parameters in Pascal) should be made more widely applicable and its use encouraged. To ensure that the programmer is fully aware of which method is being used, no default is allowed and either ref or val must be specified. In order to allow constants and read-only variables to be passed by reference, an additional option - ref read - is provided; in this case the compiler assists in detecting assignments to the parameter.

6.5 DESIGNATIONAL ASSIGNMENT [10.1.2]

The designational assignment statement is used to assign a reference to the variable on the right-hand side of the assignment to the variable on the left-hand side. The variable on the left-hand side must be a reference variable, i.e., a pointer variable, an adaptable array, a procedure variable, or a label variable. The designational assignment was judged to be a more satisfactory solution than achieving the same result by an ordinary assignment, with a built-in function returning a reference on the right-hand side.

Because of the similarity of the designational assignment statement and the 'conforms to and becomes' operator (::=), the same representation is used for both.

6.6 COMPILE-TIME FACILITIES [12.0]

There is a wide range of possible compile-time facilities which can be included in the design of a high-level language - from none at all (the most common approach), to a simple text replacement macro facility, to a PL/1-level facility which provides compile-time variables and conditional compilation, to a powerful macro facility which offers an extensive list of compile-time statements, optional and keyword macro parameters, access to the compiler's symbol table, etc.

The correct choice for SWL was not obvious. The decision hinged upon the question of whether or not macros were intended to be the sole means for handling system interfaces. If so, then an extensive set of compile-time facilities would be required, greatly increasing the complexity of the front end of the compiler and substantially decreasing compilation speed.

If, on the other hand, some other means of handling system interfaces were provided, then the additional burden of powerful compile-time facilities was felt to be unwarranted.

It was decided that until there exists a clear need for something more powerful, the compile-time facilities in SWL would be kept to a minimum. In particular, the following features are provided:

1. Compile-time variables of type integer and boolean.
2. A compile-time assignment statement for altering the value of compile-time variables.
3. A compile-time if statement to allow conditional compilation.
4. A simple, parameterized text substitution facility.

6.7 FILES AND I/O [4.2.9, 10.5]

It was decided that adequate I/O facilities must be provided for within the SWL itself, since the alternative would likely be a proliferation of incompatible I/O packages. The Pascal file variable was felt to be inadequate, however, since the components of a particular file variable must all be of the same type. The concept of a file was changed, therefore, to the more traditional one which allows access to the file only through actual transfer of information to or from a regular variable.

A basic set of sequential file operations is provided in SWL: get, put, rewind, file mode reset, write end-of-file, and end-of-file test. This set is not intended to be complete, but further extensions must wait until more is known about the IPL data management system.

6.8 MAXIMUM SET SIZE [4.2.1]

While the Pascal language specifies no limit on the number of elements that can be contained in a set, the Pascal compilers have made the restriction that a set must fit within one word of memory. Such a restriction is not made in SWL, and while there may be some implementation dependent upper bound, sets of at least 256 elements will be allowed. This accommodates the important case of set of char.

6.9 TYPE CHECKING [4.1.7, 4.1.9, 4.2.1-4.2.4]

Most Pascal operations are defined only for two operands of the same type. Previous Pascal compilers have performed this type checking in a highly restrictive way, requiring that both variables be declared with the same instance of the type definition. In SWL this requirement is relaxed substantially, and the following rules are used to determine whether two types are compatible:

1. Record types are compatible if the corresponding component types are compatible and the field selector identifiers are the same.
2. Array types are compatible if they have compatible element types, the same number of dimensions, and the same extent (but not necessarily the same bounds) for each dimension.
3. Procedure types are compatible if the types of all the parameters (including the return value, if any) are compatible. The names of the formal parameters need not agree.
4. Set types are compatible if the base types are the same scalar type.

6.10 TYPE CONVERSION [11.2.4-11.2.6]

Pascal's 'ord' and 'chr' functions allow conversion between variables of type integer and type character. It was decided that SWL should also provide for conversion between type integer and any user-defined ordinal type. To handle all these conversions in a uniform way, one may designate a conversion function from one scalar type to another by prefixing the target type with a dollar sign, e.g.,

```
type  color = (red, green, blue);
var   A: color;
      B: integer;
A := $color (2);
B := $integer (red);
```

Ordinal types are assumed to have zero-origin representation.