The SS Programming Language

Version: July 2, 2017

Table of Contents

Author and Copyright

The SS Programming Language

SS is a language designed for application programming or server scripting. It is a complete programming language, not just for scripting, designed to be compiled, not interpreted. It is a pure object-oriented language, and is designed for simplicity of use. Some typical language features in other programming languages are changed in order to achieve lower error rates in writing programs.

As a scripting language, SS is intended to be used in creating web sites and in creating tools to run under a server such as Apache. It is not intended for creation of stand-alone programs. The reason is that a server environment simplifies input and output and simplifies user interaction. There is no standard graphical library.

Unlike many scripting systems, SS is intended to be compiled, possibly in a "just-in-time" fashion, or as a more traditional compiler. The script files as a set create a library system which is managed by the compiler and the development tools.

An SS script source file has the extension ".ss" after the file name.

Source files are UTF-8 code text files, using 8-bit characters. Internal strings are the same encoding.

After compiling, the result is stored in a "library" file with extension "**.sslib**" which can be combined at run time in order to execute the script.

Features

All variables and arrays are declared and typed. It is a manifestly typed language. It is a weak typing language, meaning there are implicit type conversion rules.

Every value is an object. Constants and expressions are objects.

Inheritance is simplified and limited, with different access defaults. An interface feature is available. Function overloading is supported. Operator definition is included. It supports subtyping polymorphism. A procedure can have different implementations for different arguments.

Names in SS are significant case words The names ABC and Abc and abc are all different. There are a number of reserved words in SS. They cannot be used for names. The case of the reserved words is more flexible than other identifiers - they can be lower or upper or “Title” case.

A concern in the design is type safety. There should be fewer opportunities to confuse values, like string or array overflow, or types being misused.

Arrays and strings are designed to avoid overrun and bounds violations for safety and a degree of protection against hacking.

Unusual Features

- An array can have string indexes rather than integer indexes. Both fixed length and variable length arrays are supported. Arrays of arrays are possible. No two-dimensional arrays are allowed.
- A number of array procedures and statements are provided.
- Names can use additional European characters and accented letters as letters.
- Pointers are not used.
- Less punctuation is used than in the C family (C, C++, C#, Java, JavaScript, …). There is less use of parentheses. Some operators are different. Equality is "**=?**" rather than "**==**". This helps avoid some errors.
- The *mode* feature is different. It encourages an unusual style which avoids errors. A mode is a function based on a defined object type that implicitly returns a reference to its base object or a new object. It normally just sets an indicator or is part of the settings of an object.
- A generator function enables abstraction when obtaining a sequence of values.
- A list or tree can be made to operate like an array.
- SS has a parameterized interface and object type. Objects and functions can support multiple types.
- An enhanced object type can keep units apart, like Fahrenheit versus Celsius, or Meters versus Inches.
- The model for file manipulation is different.
- Formatting for output is designed to be less error-prone, and is not patterned after Fortran.
- Regular expression functions have an optimization feature.
- Values (variables, expressions, etc.) can be embedded in a string value, similar to PHP and PERL.
- Operators can be defined.
- Variant procedures provide an alternative to abstract procedures in an object type.
- Complex arithmetic expressions are supported.
- A large integer type 'decimal' is supported.

Omitted Language Features

SS is a “smaller” language than C++ or C# or Java in a few ways, in order to simplify implementation and the learning curve. These are some of the missing or simplified features:

- Different sizes (or precision) of floating point. One precision (double) is provided.
- Pointers and pointer arithmetic. These are omitted because of danger and misuse.
- Explicit allocation (malloc in C) or the **new** operator. Garbage collection is used automatically.
- Structs. The object includes all struct capabilities.
- Control of namespaces; modules, packages, etc. Simple scope rules are used.
- Type punning, or union. Omitted because of danger and misuse.
- Events and event handling.
- Error handling with **throw**, **try**, **catch** style. A simpler condition-handling s**ignal** and **when** are used instead.
- Threading and parallel computation.
- Prefix ++ and -- are omitted (postfix ++ and -- are included), also the C language family ternary operators '?' and '**:**' which allow a choice in the middle of an expression are omitted.
- Direct input and output from a user. Programs run on a server.
- Closures, functions as arguments, callbacks. These can be done using objects and interfaces, or variants.

Some Style Rules and Suggestions

- A script consists of object definitions and type and data definitions used in the script, or statements which identify library files.
- The main program is defined in the null based method named _Main, which is defined outside any object. It has a definition like this:

```
null method  _Main
{
  declarations and statements
}
```

- There are no executable statements, control constructs, or assignments outside of an object.
- There are no multiple statements on a line except where a control statement allows simple cases. This encourages easier modification.
- Comments follow a # sign, to the end of the line. The comment acts as an end of line.
- There are no multi-line or embedded comments.
- A statement can continue across lines if the last non-space character of a line or the last non-space character before a comment (#) is a reverse slash (\). Names and keywords and numeric constants cannot be broken. The continuation mark reverse slash is not effective if it is in a comment or string constant.
- Multi-line string constants can be written using the << and >> symbols without using continuation marks. Embedded end of line breaks are value \n.
- The end of line is either CR/LF or LF. In a character string, these are written as \r\n or \n.
- Programs are expected to use indentation for readability. The “{“ character starting a block is always on a separate line.
- A numeric constant may use an underscore as an insignificant embedded space without affecting the value.
- The SS language does not use these characters, except in string values and regular expression patterns:

  ` (grave accent)
  ; (semicolon)
  $ (dollar sign)
  ! (exclamation)

Libraries

The library inclusion statement format is: **pragma use** *library_spec*

where *library_spec* is a specifier of a library and library member, as a name list of files with assumed extension **.sslib**. There are some standard library files which are automatically included. These define standard objects or procedures. The word **use** is not reserved.

The *library_spec* has the form: "directory|member" where "directory|" can be omitted; a search is then done for the member in a standard directory/folder.

Library code must have complete object or other definitions. Abstract definitions are allowed.

Terminology

The word “object” in this language and this language description is used in an atypical way. Most object-oriented programming languages (OOPLs) use the word class instead as the keyword for the definition of a new object type.

In SS, the terms which may need to be clarified are:

- ***object*** - the traditional object, an encapsulated thing containing data and associated properties and methods. In SS, every data item or value is an object, even the basic things like constants and variables. These implicit objects have predefined properties.
- ***object type*** - the new type which is defined by the construct beginning with the word **object**. This is actually a class in common programming language terminology.
- ***enhanced object type*** - an object definition can inherit from a basic type, such as float, and enhance the meaning of that type. This allows a way to add attributes or restrict usage.
- ***default object*** - the language assumes everything is an object, with some built-in properties and procedures. An object that has base_type **default** and has no name is a way to add baseless procedures and property items.
- ***member*** - every item contained in an object is a *member* of that object. Also a single value in an array is an array member.
- ***index*** - the subscript of an array. It can be a positive int, uint, nameset or string.
- ***statement*** - a line or construct which defines an action.
- ***construct*** - a multi-line statement (if, for, final, switch, when) or declaration (a procedure definition, object type, or interface.)
- ***declaration*** - a line or construct which defines a type or defines a procedure. This is not an executable statement.
- ***block*** - a sequence of statements enclosed in a pair of curly braces. A block is a statement. Also a ***case block*** which is not a statement.
- ***procedure*** - a subroutine (a “method”) or function or mode, defined inside an object. In some languages these concepts may be called methods or messages.
- ***method*** - a procedure or subroutine which does not return a value, thus it cannot be used as a primary element of an expression. It is used as a statement.
- ***function*** - a procedure which returns a value, usable in an expression.
- ***mode*** - a special kind of function. A mode:
  - is based on a given object type,
  - usually sets or changes the state or status or values in the base object,
  - always returns a reference to the base object or another object as an answer,
  - is usable as a base object.

- ***selector*** - the period character ("**.**") - the left side is an object or a type, the right side is a member, type name, built-in procedure and many other uses.
- ***accessibility*** or ***access*** - the level of access to a member of an object. One of these levels applies:
  - ***private*** - visible and accessible only in the object and its members, not in inheriting objects. Defined with the option **private**.
  - ***protected*** - default, not specified with a keyword. Accessible to members and within inheriting objects.
  - ***public*** - accessible anywhere. Defined with the option **public**.
- ***property*** - not a variable, but it appears as one. This is a means of controlling access to data, and also a way to apply a conversion or a way to hide the properties of data. A property item may invoke a procedure or use an expression or assignment to implement a **set** or a **return** phrase or both.
- ***named constant*** - a name and type can be assigned a constant value.
- ***Lvalue*** - an item which can be assigned a value in an assignment. A 'left-hand' value.
- ***reference*** - argument or base object passed without copying the value, using a pointer.
- ***value type*** - a type which is passed by copy. The types uint, int, logical, float and nameset.

Reserved Words

Reserved words cannot be used to define names for a program; they have fixed uses in SS.

Reserved words can be lower case, upper case or with an initial capital letter, “Title” case. These are the same: **case, CASE, Case**. The choice of case used is left as a style issue. Other mixed case forms are not reserved words. CaSe is not reserved.

An exception is for the four basic type names beginning with **U** (meaning “unsigned”**)**, which also allows the forms **UByte**, **UInt**, **ULong** and **UShort**.

Reserved words are also called keywords. There are 58 reserved words. Counting all allowed case variations, there are 178 reserved words.

Table: Reserved Words

abstract	**decimal**	**float**	**method**	**private**	**static**	**ushort**
and	**default**	**for**	**mode**	**public**	**string**	**variant**
bit	**each**	**function**	**nameset**	**ref**	**switch**	**when**
byte	**else**	**if**	**not**	**repeat**	**this**	**while**
break	**enables**	**inherits**	**null**	**return**	**true**	
case	**end**	**int**	**object**	**self**	**ubyte**	
complex	**exit**	**interface**	**or**	**set**	**uint**	
const	**false**	**logical**	**parent**	**short**	**ulong**	
continue	**final**	**long**	**pragma**	**signal**	**until**	

Names

A name must begin with a letter or an underscore or the at-sign @, and may use either case of letters in the name. Underscores and at-signs are allowed anywhere in a name. Digits are allowed after the initial letter or underscore or at-sign.

Initial underscore is reserved for system-defined names.

Letters are the 26 Latin (and English) letters A through Z, upper or lower case and Western European letters and accented Latin letters as follows:

lower: à á â ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ
upper: À Á Â Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ
additional lower: ß ÿ

Legal names: inTheMiddle, a1, a_name, time_worked, arrêté, time@work.

Invalid names: 1A, two words, A&B, x-hyphen.

Scope of Names

A name has a scope limited to the object it is in (unless an access modifier is specified), or to the block in which it is defined. The scope begins where the name is defined. At the end of that block, the scope is terminated and the variable, array, string, or object is erased.

All names are defined in an object. Those in the default object have global scope.

Names of object types, interface names are in the same category and a name must be unique for those usages. They all act as type names.

Data Types Supported

The standard data types in SS are integer, float, logical, digital, complex and string. They are also called basic types.

New types can be added as objects. Types complex and digital use built-in object definitions.

Integer

Integers have no fractional part, and are (by default) signed 4-byte integers, with type keyword **int**. Integers of other sizes can be declared. They are declared as **int.64** (also called **long**), **int.32** (the same as **int**), **int.16** (also called **short**) and **int.8** (also called **byte**). All of these are signed. The appended number is the number of bits used.

Unsigned integers are also available, as **uint.64** (also called **ulong**), **uint.32** (also called **uint**), **uint.16** (also called **ushort**), **uint.8** (also called **ubyte**) and **uint.1** (also called **bit**) types. The size numbers are bit lengths.

The alternate names can be used interchangeably. Both **uint** and **int** types are integers.

The integer value ranges (shown with commas) are:

uint.1 0 or 1 (also called **bit**)
uint.8 0 to 255
uint.16 0 to 65,535
uint.32 0 to 4,294,967,295 (maximum for **uint**)
uint.64 0 to 18,446,744,073,709,551,615

int.1 0 or 1 (unsigned)
int.8 -128 to 127
int.16 -32,768 to 32,767
int.32 -2,147,483,648 to 2,147,483,647 (default size for **int**)
int.64 -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Float

Values of type **float** are based on the IEEE 754-2008 standard, implementing double precision binary floating point arithmetic, using 8 bytes. There is no short or long (double or quadruple) form. The number of significant digits is about 18.

A float constant is a floating point value that consists of a sign (default is plus), a numeric value where each digit in the value is from 0 to 9, a required decimal point, a required digit on each side of the decimal point, and an optional exponent after a letter E (or e) that indicates a multiplier by a power of 10. The type keyword is **float**.

Logical

The result of a comparison or logical expression is either **true** or **false**. A logical value occupies 1 byte in storage. An array of logical values packs the values into bits.

The **if** statement and other places require a logical expression. The arithmetic (non-complex) types may be type cast to logical. A nonzero value becomes **true**, a zero or null is **false**.

String

Strings are varying in length. There is no maximum length. The string type in SS is not terminated by a "null character" as in C/C++. Characters in a string are in the UTF-8 code, using one or more 8-bit bytes. There is no separate type for single characters. Strings are immutable; internal values cannot be modified.

A *pattern* is a compiled and verified regular expression pattern string. A pattern constant begins and ends with a slash rather than either apostrophe or quote marks. A string expression can be a pattern and it will be compiled at run time. This is a special form of string, not a type.

Complex

Complex type is a pair of numbers, one is real and one imaginary. The type of the interior numbers is **float**. Complex variables are immutable; you cannot alter part of the value.

The complex type is implemented internally by treating each reference to **complex** (or **Complex** or **COMPLEX**) as a reference to an object named **_Complex** which is found in the library **complex.sslib**. This allows alternative implementations to be defined.

Decimal

Decimal type is a very long signed integer. The current implementaion is up to 36 digits. Decimal variables are immutable; you cannot alter part of the value.

The decimal type is implemented internally by treating each reference to **decimal** (or **Decimal** or **DECIMAL**) as a reference to an object named **_Decimal** which is found in the library **decimal.sslib**. This allows alternative implementations to be defined.

Nameset

Nameset items are named constant unsigned integer values, grouped into a name. This is similar to the C language 'enum', but more restrictive because the names are typed, somewhat like C language '**typedef enum**'. A nameset constant can be used only with a value or array using that nameset.

Value Types and Reference Types

Value types are simpler data types which pass a copy of a value to a procedure, and which copy a value when they are assigned. These are the builtin types except digital and complex.

Reference types are more complicated. When they are passed to a procedure, a copy of a reference (a hidden pointer) is passed. When an assignment is done, a reference is assigned, rather than the value. These are arrays, strings and typed objects.

Two references can refer to the same copy of data. If one item is altered, both references see the same altered data.

Typed objects provide some control over this behavior, also procedures may force a certain copy or reference behavior on their arguments.

Operators

Table: Operators in Expressions

Operator	Description	Priority	Associativity	Operand Types
++	Postfix incrementation	11	n/a	int
--	Postfix decrementation	11	n/a	int
.	Member/procedure selector	10	left	(any)
+	Unary +	9	n/a	int float complex decimal
-	Unary -	9	n/a	int float complex decimal
~	Unary bitwise complement	9	n/a	bit ubyte uint ulong
^	Exponentiation	8	right	int float complex decimal
*	Multiplication	7	left	int float complex decimal
/	Division	7	left	int float complex decimal
%	Remainder / Modulus	7	left	int uint decimal
&	Bitwise and	7	left	bit ubyte uint ulong
<*	Bitwise left shift	7	left	ubyte uint ulong
*>	Bitwise right shift	7	left	ubyte uint ulong
+	Addition	6	left	int float complex decimal
-	Subtraction	6	left	int float complex decimal
|	Concatenation of strings	6	left	string
~	Bitwise inclusive or	6	left	bit ubyte uint ulong
=?	Bitwise exclusive or	6	left	bit ubyte uint ulong
\=	Equality comparison	5	left	(any)
<	Not equal	5	left	(any)
<=	Less than	5	left	int float string decimal
>	Less or equal	5	left	int float string decimal
>=	Greater than	5	left	int float string decimal
not *or* \	Greater or equal	5	left	int float string decimal
and *or* &&	Logical NOT (unary prefix)	4	n/a	logical
or *or* ||	Logical AND	3	left	logical
=	Logical OR	2	left	logical
op=	Assignment	1	right	[special rules]
.=	Same as L = L op R	1	right	[special rules]
<>	Function assignment	1	right	[special rules]
	Swap	none	n/a	(any, same)

Highest priority numbers are evaluated first.

The assignment operator and the *op*= assignment have a value, the same as the left operand.

They associate right-to-left, which permits assignment to multiple variables:

abc += ijk = xyz = 123

Operator assignment *op*= is an operator (+, -, *, /, <*, *>, %, &, |, ^, ~, .) followed by an equal sign, with no space between. For a given operator *op*, the statement

a *op*= x

is the same as:

a = a *op* x

where “a” must be an Lvalue or an array.

Use of the “**.=**” operator requires the right side to be a function or return-type property name. The left side is an Lvalue, the right function or property is applied to the left side.

Examples:

```
float xyz
xyz .= abs      # if xyz is negative, make it positive
int list[] = int[3, 9, 7, 1, 0, 4]  # define an array
list .= sort       # sort the list
```

The postfix operators ++ and -- return the left operand as value, then increment or decrement it by 1. Thus, if mm has the value 4, mm++ has the value 4 but changes mm to 5. This operator as a statement also acts as an assignment statement.

The exponentiation operator **^** is right associative; the expression **a^2^3** is the same as **a^(2^3)**. The exponent must be a small int, -32 to +32.

The swap operator (**<>**) expects the right side to also qualify as an Lvalue, a left side. The values of the two sides are exchanged. Both sides must be the same type. This avoids having to use a temporary variable and three assignments.

The comparison operators (**=?**, **\=**, **<**, **<=**, **>**, **>=**) return a logical value **true** or **false**. The values **true** and **false** cannot be compared with anything. Thus, the expression **a < b < c** is invalid.

The remainder operator **%** returns the remainder from integer division. For **z = x % y**, the sign of the remainder z is the sign of x. For any int or uint the result z is z = x - (x / y) * y, using integer division.

Shortcut Operators AND and OR

The operators AND and OR evaluate the right side only as necessary. The left side determines the value alone for some cases:

Table: AND and OR Operators

Left Side	Operator	Right Side	Result
TRUE	OR	skipped, not evaluated	left side alone determines value is **true**
FALSE	OR	evaluated	the value is the right side value
TRUE	AND	evaluated	the value is the right side value
FALSE	AND	skipped, not evaluated	left side alone determines value is **false**

This skipping is called "short circuiting" or “shortcut.” Any functions or modes or postfix ++ or -- in the right side which were skipped may have had side effects which will not be done.

The bit-wise operators **&** and **|** and **~** do not short circuit.

Postfix Operators ++ and --

Postfix operators ++ and -- require the left operand to be an Lvalue. An object reference to an element completes the reference, then increments it. For an object member, the left side then must be public or a property name with both **set** and **return**.

There are no prefix versions of these operators.

These can be statements because they alter a value.

Variables

A variable is a single value of any type, declared:

type name_list

Example:

```
int abc
string name, last_name, nick_name
```

It has a *type*, which is **int**, **uint**, **string**, **float**, **digital**, **complex**, or an object type name.

An initial value can be assigned, any known values, not just constants:

int maxSize = 1000

Array declarations can be mixed with variable declarations:

float xyz, totals[100], data[]

Within a declaration, only one type can be named. This would be invalid usage:

int ijk, string str[]

Implicit Type Conversions

An assignment will convert the type of the right side to the type of the left side if it can do so.

Unsigned **uint** widens as it converts to int. In other words, **uint.16** widens to **int.32**.

Conversions to and from the numeric types int and float are obvious. Overflow (loss of significant digits) in converting from float to int is ignored.

A logical value can be cast as an integer or a string. When converted into an integer, **false** is 0, **true** is 1. On conversion to a string, the values are 'T' and 'F'.

Conversion from a numeric to string is allowed, and is the same result used in the standard methods print and printLn, or the same as a value insertion in a string. Integers are exact, float numbers are approximate.

Conversion from a string to a numeric value may cause an error if there is no valid conversion after trimming off blanks. An empty string converts to 0, and a value too large to be an int is treated as a float type.

Unlike some systems, a string value like "123GO" does not convert to the integer 123.

An **int or uint** or any length variant will convert automatically to a **float** in an expression where a **float** is required.

Integer Arithmetic

Integers (32, 16 bit, 8 bit or 1 bit) "widen" to the next size in an expression, when combined with +, -, or * operators. There is no widening for division or for the bit-wise operators. 64 bit values do not widen.

When a longer integer is assigned to a shorter integer, the upper part is discarded without error; Integer overflow or truncation is not diagnosed.

Widening or shortening can be specified by type cast; abc**.int.8** extracts the low 8 bits of the integer abc.

Nothing automatically widens to decimal type.

Logical Expressions

A logical expression has a value of **true** or **false**. These are also reserved words for the values. Arithmetic on these values is disallowed; the value of **true** is nonzero but not specified. A non-logical value cannot be compared with **true** or **false**. Logical values can be compared with =? or \=.

An integer or float value which it is cast to logical is considered to have the logical value **true** if it is nonzero, or **false** if it is zero.

A logical expression can be assigned into an integer variable or array element; the value **false** will be 0 and the value **true** will be 1 when assigned.

Null

The value **null** is a constant. It is the state of an array or object that has no instance assigned, and it can be returned from a function which returns an array or object, to indicate instantiation failure.

The **isNull** or **isNotNull** properties can be used to check for **null**. A value cannot be compared with the constant **null**.

A **null** value is treated as **false** in a logical context.

Type Casting

A value can be converted to a compatible type by casting the type:

value.*type_name*

converts the type or changes to a compatible type. If the value can be assigned to an item of the *type_name* specified, it is a legal type cast. This is not "type punning" which retypes without conversion. There is no type punning in SS.

An object variable can be cast as a type it inherits. The result loses any additional features and makes a copy.

A value that has a type cast is considered a new value, so a reference to the old value is not used.

Type casting applied to an array applies to each element, creating a new array.

Any basic type casts to string. A string casts to a type if it can be legally converted.

Table: Legality of Non-String Type Casts

base \ cast	uint	int	float	complex	logical	decimal
uint	Yes	#1	Yes	Yes	#2	Yes
int	#1	Yes	Yes	Yes	#2	Yes
float	#3	#3	Yes	Yes	No	No
complex	No	No	No	Yes	No	No
logical	No	No	No	No	Yes	No
decimal	Yes	Yes	No	No	No	Yes

#1: **int** and **uint** - the sign may change to a data value, and vice versa; data loss can occur if the size of an integer value is shortened. Integer sizes remain part of the type; **abc.uint.8** extracts the low-order byte.

#2: The logical value is **true** if the base is nonzero, **false** if it is zero or null.

#3: The integer part is used; truncation is an error.

Data Values

Named data can be any one of these things, all of which are considered objects:

- A simple item with a type float, int, uint, digital, complex or string.
- An array, which has simple integer indexes or string values as indexes. The first index is 0 when unsigned integer or nameset indexes are used. An array element can be missing, or undefined. A missing numeric value has the value 0, and a missing string value is the zero-length string value '' (two apostrophes.) Missing objects are **null** references. Array elements of a reference type are references.
- An object, which may have procedures in it. The definitions may be marked **private** or **public** to control access. The type is defined with **object**, and variable or array names or functions can be assigned to that type. An object can inherit, and can enable an implementation.
- A named constant.

Constants

Integers (**int** or **int.32**) are signed integer values between -2147483648 and 2147483647. The sign is not part of a constant, it is a unary operator + or - in an expression. Integers use four bytes. A 16 bit integer constant has a suffix of s16 or S16, and an 8 bit integer has a suffix of s8 or S8. An unsigned constant has a suffix of u or U and a length can be shown as u16 or u8.

An unsigned or signed integer constant can also be written as 0x*dddddddd* where each *d* is a hexadecimal digit. The 'x' and the hexadecimal digits A through F can be either case. There are up to 8 hexadecimal digits in a **uint.32**; there are 2 hex digits per byte . Length and sign suffixes with **u** or **s** are optional. Otherwise it is assumed unsigned. 0x80000000 is the largest integer negative value, and 0xFFs8, 0XabcdU16, 0xCD are examples.

There is no support for octal values. A leading zero is ignored in a constant. (This is a difference from C or C++.)

A binary constant can be used for signed or unsigned integers. It has the form 0**b**ddd...d or 0**B**ddd...d where d is a binary digit (bit) of value 0 or 1. A sign or unsigned suffix (**s** or **u**) with optional bit size can be on the end. By default, a binary constant is unsigned. The value has extra leading 0 bits added at the left (high end) as needed. At most 64 bits are allowed.

A decimal constant begins with 0d or 0D followed by the digits. Currently, up to 36 digits are allowed. Underscores are allowed for spacing.

Float constants have a decimal point and an optional exponent. A digit must precede the decimal point.

A string constant is enclosed with apostrophes, or quotation marks, slashes, or << and >>. If it begins with apostrophe, an embedded apostrophe must be preceded by a reverse slash. Similarly a quotation mark inside a string constant started with a quotation mark must have a preceding reverse slash.

A string constant delimited by a slash is assumed to be a regular expression pattern, and is checked and "compiled" at compile time. This is legal only where a pattern value is used.

Any numeric constant may have underscore characters embedded anywhere for readability, without affecting the value. This may help in counting digits for long values.

An array constant is written as a list of values with surrounding square brackets after a type name. The type of all members must be the same. The type of the indexes must be the same. A final comma is allowed in the list, but no value is assumed after it. Index values may be supplied in the list.

A nameset constant is written like an array constant, as a list of the nameset values with surrounding square brackets after a nameset name. All members must be from the nameset. A final comma is allowed in the list, but no value is assumed after it.

Arrays

An array is declared with an *array_spec* after the *name* as follows:

[[*index_type* | *nnn*] **]**

This is optionally followed by an initial value:

[**=** *init_val_expr*]

There are no multiple dimensional, or "square" arrays. An array of arrays is permitted, by repeating the *array_spec* part, in square brackets, after the *name*. An array of arrays can be called a *jagged* array because the array lengths can vary.

The preceding *type_name* defines the type for all array elements.

The *index_type* can be int or uint or string or a nameset type name. Default *index_type* is **uint**.

IAn array index cannot be type **float**, **logical** or an **object**, including **complex** or **decimal**.

The optional *nnn* implies a fixed size array of *nnn* members, where *nnn* is a nonzero positive integer constant or integer expression. A fixed size array is permitted only when the index_type is uint, not for a nameset or string index type. An index value must be less than *nnn* when the array is fixed size. An integer index is never negative for a fixed size array. The index begins at 0 for a uint or int index.

Examples:

```
string arr[]
int iarr[]
float cost[string]
```

An array has integer indexes, or string valuesas the index. When the index is uint, it starts with 0 for the first element The maximum index of an array is unspecified unless the array is fixed size, which has a maximum index 1 less than the size nnn, or it is indexed by a nameset.

Example:

```
uint msiz=100
float scores[msiz]
int cards[52]
string table[10][string]  # an array, 10 members, of string-indexed arrays of strings
table['nick'][5] = 'pinky'
```

Initially an array declared as shown has no elements. An empty array value is written as *type*[]. *Index-type* if omitted is uint. Values can be assigned to single or multiple elements as shown:

```
int arr[MSIZ] = int[100, 25, -6, 94]  # where MSIZ is 100
arr[6] = 'XYZ'      # skipping indexes 4 and 5, remaining unassigned
arr[MSIZ-1] = 0xAA  # into array element 99
```

The index values (if int or uint type) in the [] initializer can be omitted or specified before the value, with an equal sign:

```
int arr[] = [23,-88,10=123,345]  # defines four array elements with indexes 0,1,10,11
```

An array that is indexed with string values as indexes is implicitly sorted by key value.

Values can be assigned to single or multiple elements as shown:

```
float salary[string] = ['driver'=12000.00, 'sales'=15000.00]
salary['CEO'] = 149995.00  # adds a new element
```

An array of arrays can have an initial value:

```
string class[string][] = ['English'=['Walt Whitperson','Will Shakefist'], \
        'Geometry'=['Archie Medes','Ima Euclidian','Neva Meetin'], \
        'Sports'=['Wilt Nicklaws']]
```

For an array of arrays, a reference with one index is a reference to an array.

An array is not instantiated unless it is either declared as a fixed length array, or is assigned an initial value. Declaring a non-fixed-length array establishes a place-holder for a reference to an array; it is not instantiated until an array value is assigned or a reference to another array is assigned.

A non-fixed array is implemented as 'map', as a tree structure, like the _BST described in an appendix. A non-fixed array with **int** indexes can have negative index values.

A reference like Arr[] as a left value (an Lvalue) in an assignment, where Arr has integer indexes, is assumed to create an index value which is 1 higher than the maximum index value, or 0 if the array Arr is empty. This is invalid for indexes other than **int** or **uint**, and for a fixed-size array.

Array Elements

The array elements are indicated as follows:

```
arr[0]   arr[1]   arr[n]    # array arr
Karr['start']   Karr['last']     # array Karr
arr[i+1]
```

An array with nameset values as indexes uses its array index like a variable or named constant.:

```
partNum[CAM] = 1234
```

A missing element reference in an expression returns **null**. Existence can be checked with **isNull** or **isNotNull**.

Array Expressions

An array name passed as an argument to a procedure is a reference to the array, not a copy.

Assignment to an array name is interpreted differently in these circumstances:

- abc = def # both are arrays of the same type; copies the reference. Both arrays refer to the same data.
- abc = def # where the types differ, but can be converted, builds a new array and assigns a reference to it.
- abc = def.copy # assigns a ref to the new copy of def. They refer to different data sets.
- abc =? def # is true if the two arrays reference the same data and dimensionality and indexes. Inequality also is allowed.
- abc *op*= *val* # applies the operator *op* with the scalar *val* to each element.
- abc *op* def # returns an array, applying the operator *op* on elements of the two arrays with matching indexes. If for one array a value is missing, the missing value takes a default of 0 or the empty string. Multiply (*) is not matrix multiply. The arrays are treated as vectors. The operator must be valid for the type of the array element values.
- Bit-wise operators can be applied to an array of integers.
- abc *op*= def # equivalent to: abc = abc *op* def, where def is an array or a scalar value.
- abc **.**= *proc* # where *proc* is a property or function, applies *proc* to each member of abc or to the whole abc if *proc* is an array-based procedure.
- abc.proc # also applies method proc to each member, or produces a new array if proc is a function invocation, applied to each member, or applies proc to the whole array if it is an array procedure.
- A procedure can be applied to an array. Examples:

```
int cnt = iarr.Count   # gets the number of elements in array iarr
arr = arr.sort         # sorts the values in an array with integer indexes
arr .= sort            # an alternate way to do the same sort
a_array.Clear          # removes all elements
```

Array Functions, Properties and Methods

Table: Built-in Array Functions, Properties and Methods

Name:	Returns:	Notes:
allZero	logical, **true** if all members are zero, else **false**	
anyNonZero	logical, **true** if any member is nonzero, else **false**	
Common(b[])	all non-null values which also exist in **b**. Base & argument **b** are the same type. String indexes must match	
Clear	method, empty array. Delete element. Reset string to empty	S
Count	the number of elements, or length of string in characters	S
Find(key)	the member that first matches key as index, else **null** if not there	S
First	lowest index in the array	
Flip	exchanges indexes and values of an array. Duplicates are lost	
indexType	string – value is 'fixed', 'mapped', or 'nameset'	
Join(string sep)	string, concatenated with separators sep, on a string valued array	
Keys	array of indexes for used elements of an array	
Last	the highest index of an array	
Max	maximum value of an array	
Merge(b[])	array with elements added from **b**. If fixed size or integer indexes, merged by value else merged by key. Duplicate keys replace the value from **b**	
Min	minimum value of an array	
namesetArray	if nameset indexed, array of names and values, string indexed else **null**	
namesetName	nameset name (string) else ''	
Pop	return last element of non-fixed integer-indexed array and remove it	
Push(val[])	adds new last elements on a non-fixed array with integer indexes. Type of array val is same as base type. Adds all elements from val	
Remove(b[])	array with elements removed that match indexes of array **b**.	
Slice(int m, int n)	an array/string extracted from index **m** through **n**, **m** <= **n**	S
Sum	the sum of the elements of an arithmetic value array, zero if empty	
Unique	array with duplicate values removed, keeping lower of the indexes	
Values	array of values ignoring the indexes. Integer indexed result	

In the above table, S in the Notes column means a string variable is supported as an array.

Built-in array procedures are defined in standard library member **array.sslib**.

String as an Array

A string value is partially treatable as an array of one-character strings. The characters are indexed from 0. Indexing a string is not allowed in an Lvalue since strings are immutable. The array functions which apply are limited as indicated in the above table.

A 'character' may occupy several bytes.

Nameset Feature

A defined data type can be introduced with the **nameset** declaration:

nameset *nameset_name* name [= integer_value] [**,** ...]

The names are names for a set of values, possibly with a constant value for the name after an equal sign. If no integer value is shown, the first name is assigned the value 0, the next is 1, etc. Once a value is assigned to a name, the next value is assumed to be 1 higher.

For each nameset, the names defined must be unique, not the same as a variable or other item, and no two names will have the same value. No overlapped values are allowed. Arithmetic on nameset values is not allowed.

Two uses are possible for a nameset and its list of named values.

Nameset Values in a Variable or Array

The first is to declare that a uint variable can be assigned values from the list. The variable can then be tested for equality with a name from the nameset, or the names can be used in a switch or a **for** statement. The maximum value is 4,294,967,295 – the maximum for a **uint.**

This is declared by showing the nameset as if it is an initial value:

```
nameset JobTitles manager, souschef, chef, waiter, busboy
uint.8 job =  JobTitles
```

The value names cannot be assigned into a variable using a different nameset type. Conflicting names can be resolved using a type cast notation.

Items in a nameset variable cannot be compared to integer values. An empty or uninitialized nameset data value has a value of 0, which may or may not match a value.

For example:

```
# assigns unknown = 0, bicycle = 1, auto = 5, truck = 6:
nameset vehicles unknown, bicycle, auto=5, truck
# declare an array:
uint rigs[] = vehicles # takes vehicle values only, integer indexes
rigs = vehicles[auto,truck,truck,auto] # note the array constant uses the nameset name
rigs[2] = bicycle
# show conflicts - - -
nameset Tstat closed, reading = 8, writing, open = 15 # values are 0, 8, 9, 15
nameset DoorStats open, ajar, closed # note the name conflicts
iint frontDoor =  DoorStats
frontDoor = ajar            # legal
frontDoor = closed          # invalid, conflicts with Tstat nameset above
frontDoor = closed.DoorStats   # this is legal, using a cast to disambiguate
```

Nameset as an Array Index Type

The second use for nameset allows the named constants to be indexes in an array with that nameset type as the index type. This provides an easy way to identify bits in a logical array, for example, as a 'mask" or 'flags' item. It also allows a space-saving, more efficient compression to avoid using strings as an index for an array.

The maximum value for a nameset used as an index depends on the array type. For a logical array the maximum value is 1023. All other types the maximum index is 4,294,967,295 - the maximum for a **uint**.

Once declared, array members can be referred to in two notations – indexed by the names or as a pseudo object member.

Example:

```
    nameset TrainingReceived coder, site_designer, graphics, font_design, writer, editor
    logical training[TrainingReceived]
    training.writer = True
    training[editor] = True
.   training.site_designer = True
    # or, this is equivalent - - -
.   training =  TrainingReceived[editor,writer,site_designer] # clears other flags
.
```

Information About a Nameset

Every nameset implicitly creates a static-like read-only array named **Array** with string indexes and integer values of the associated constants. This array cannot be modified. Existing procedures can be applied to the array. For example, the nameset **vehicles** above creates the array **vehicles.Array** with string keys and values:

```
vehicles['auto'=5,'bicycle'=1,'truck'=6,'unknown'=0].
```

Array properties like **Count** and **Max** can be determined by **vehicles.Array.Count** (4) and **vehicles.Array.Max** (6). To save time, these two values are also available as **vehicles.Count** and **vehicles.Max**.

Strings

A string is an object which resembles an array of characters. Strings are immutable. In SS each character is composed of one or more 8-bit bytes in the UTF-8 code. The characters (not bytes) in a string are numbered from 0. There is no special character value that terminates a string. Strings are not fixed length. There is no separate "character" type.

String constants are enclosed in apostrophes (') or in quotation marks (“). The difference is that a string constant enclosed in quotation marks is scanned for values to be inserted, and it permits escaped character sequences as listed in the table below.

When a string constant is enclosed in quotation marks (but not apostrophes), variables, array elements and even expressions that can evaluate to a string can be embedded by enclosing them in curly braces. Spaces in the insert are ignored. If a left curly brace is intended to be a character, precede it with a reverse slash.

Example:

```
string drink = 'tea', msg = “A cup of {drink}.\n”
```

A string constant delimited by ' or " must end on the same line; it cannot be continued. If it begins with apostrophe, an embedded apostrophe must be preceded by a reverse slash. Similarly a quotation mark inside a string constant started with a quotation mark must have a preceding reverse slash.

A string constant can also be delimited by << and >>. The << must end a line or be followed by optional spaces and a comment and the >> must be in the left two positions of a following line. Ends of lines except the line with << are included as '\n'. Inside the string constant. Insertions are allowed as in a string constant that starts with a quotation mark.

Example:

```
string text = << # two lines:
this is line 1
this is line 2 with insert {abc}
>>
```

A regular expression pattern string is written with enclosing slash (/) marks.

String values are concatenated with a plus sign operator. Concatenation is not implemented inside a string constant except in an expression enclosed in curly braces.

Like arrays and defined objects, strings are a reference type. Strings are immutable.

A string's length is limited to a value that fits in a uint.32 type variable, a bit over 4 billion characters.

A string can be indexed like an array to retrieve a single character:

```
string sv = 'abcdefghi'
print sv[3]   # outputs 'd'
```

Table: Escaped Characters in a String

Escape Chars:	Meaning:
\n	End of line
\r	Carriage return
\f	Line feed character
\t	Tab character
\\	Reverse slash
\{	Left curly brace
\xdd or \Xdd	Hexadecimal value dd (exactly 2 hex digits)

There is no support for the BELL or VERTICAL TAB or FORM FEED codes or other holdovers from teletype days.

There are no decimal or octal or binary character codes, only hexadecimal.

A reverse slash before any other character is retained.

String Expressions

Concatenation example:

```
string drink = 'tea'
string tasty = 'iced ' + drink
```

A string acts like a array of characters when a subscript is used. The character referenced is extracted as a substring of one character. Thus, *stringvar*[*n*] identifies the single character number **n** of *string*. This substring cannot be assigned a value because strings are immutable. It cannot be on the left of an assignment.

The characters of a string are numbered from 0, like arrays.

Assignment copies a reference, like an array.

Strings can be compared with the comparison operators =?, >, >=, <, <=, \=.

To be equal, the reference is the same, or the length and all character values are the same.

Substrings are string values extracted from a string. The built-in function substr extracts a substring:

```
string a = 'abcdefghijkl'
print a.substr(3,5)  # prints "defgh"
```

Table: Built-in String Functions and Properties

Function:	Description:
after(string s)	Returns substring found after substring **s,** or **null** if none
before(string s)	Returns substring from beginning up to substring **s, null** if none
between(string s1,string s2)	Returns substring after **s1** and before **s2**, or **null** if none.
byteCount	The number of bytes representing the characters
copy	Returns a copy of the string
Count	Length of the string – the number of characters
findLeft(string s)	Integer position of the first substring **s**, -1 if not found
findRight(string s)	Integer position of the rightmost substring **s**, or -1
from(uint m)	Returns substring from character **m** up to the end, '' if none
lastn(uint n)	String returned, length **n**, from the end
lower	String returned with all letters (with accented) lower case
replace(string patn,string s)	String returned with all **patn** substrings replaced with **s**
Slice(uint m, uint n)	Returns substring, characters numbered **m** through **n, m** <= **n**
split(string sep)	Returns array of strings where **sep** is a separator
substr(uint m, uint n)	Returns substring from character number **m** for **n** chars
title	String returned has the first letter upper case, the rest lower, for each word
trim	String returned with leading and trailing spaces removed
trimLeft	String returned with leading spaces removed
trimRight	String returned with trailing spaces removed
upper	String returned with all letters (with accented) upper case
upto(uint m)	Returns substring from character 0 up to (but not including) character **m** (the first **m** characters), **null** if none or if **m** is less than 1

The built-in string function prototypes are defined in the standard library member **string.sslib**. Note that copy, Count and Slice are also array functions.

Accented letters may occupy two or more bytes in the UTF-8 code as a single 'character'.

Trim functions remove spaces and \n, \r and \t (Tab) codes

The upper and lower functions also support the accented letters which have two cases in the UTF-8 code (see Names section, above): **"á".upper** yields **"Á"**.

Examples:

```
string s = 'abcdefghij', ss
int pos = s.findLeft('def')    # set to 3
ss = s.before(8)   # truncates to 8 characters
ss = s.from(3).before(4)   # gets 'defg' substring
string txt = 'apple, peanut, cheese'
# set array with 3 strings: 'apple', 'peanut', 'cheese' - - -
string words[] = txt.split(', ')
words .= trim  # trim each of the values
```

Mathematical Functions and Properties

Table: Mathematical Functions and Properties

Fun(y):	Base(x) Type:	Result Type:	Returns:
abs	float int complex decimal	(base)	Absolute value
arccos	float complex	(base)	Arc Cosine
arg	complex	complex	Argument, theta, angle for polar
arcsin	float complex	(base)	Arc Sine
arctan	float complex	(base)	Arc Tangent
arctan(**y**)	float	(base)	Arc Tangent of **y**/**x,** y is float
binary	(any uint or int)	string	Bits in the integer as a string
bitSize	(any uint or int)	uint	Returns 1, 8, 16, 32 or 64
ceil	float	(base)	Whole number > or =, away from 0
conj	complex	complex	Complex conjugate
cos	float complex	(base)	Cosine (of radians)
cosh	float complex	(base)	Hyperbolic Cosine
exp	float complex	(base)	Exponential, **e** to the power
floor	float	(base)	Whole number, truncated toward 0
hex	(any uint or int)	string	To a hex string, no leading 0x
isInfinity	float	logical	Test for "Infinity"
isNaN	float	logical	Test for "Not a Number"
isNeg	int float decimal	logical	Returns true if < 0 else false
isNonZero	int float complex decimal	logical	Returns true if \= 0 else false
isZero	int float complex decimal	logical	Returns true if =? 0 else false
log	float complex	(base)	Logarithm, base ***e***
log10	float complex	(base)	Logarithm, base **10**
Maximum(**y**)	float int decimal	(base/y)	Maximum of base(**x**) and **y**
Minimum(**y**)	float int decimal	(base/y)	Minimum of base(**x**) and **y**
Reciprocal	complex	(base)	Complex 1/x
round(uint **n**)	float	(base)	Round **n** digits away from zero
roundEven(uint **n**)	float	(base)	Banker's rounding, even lowest digit
sin	float complex	(base)	Sine (of radians)
sinh	float complex	(base)	Hyperbolic Sine
sqrt	float complex	(base)	Square root
tan	float complex	(base)	Tangent (of radians)
tanh	float complex	(base)	Hyperbolic Tangent
TimesI	complex	complex	Multiply by the imaginary ***i***

The standard math functions are defined in standard library member **math.sslib**. Complex functions are defined in **complex.sslib**, decimal functions in **decimal.sslib**.

In addition, the identifier _MATH_PI is defined as a float named constant with the value pi (π) to full float value in the automatically included math library. This identifier can be redefined - it is not reserved. Same for _MATH_E as constant ***e*** and other values.

Procedures: Functions, Methods and Modes

A *function* returns a value, and thus can be used in an expression. It is invoked by naming an object on which it is defined, followed by a selector period, then the function name, then arguments in parentheses, if any. The object is an implicit argument to the function.

A *method* does not return a value It is invoked like a function, except that no parentheses are used around arguments.

A *mode* is a procedure which implicitly returns a reference to an object, usually the base object. It must be based on an object. A mode may have no arguments.

Functions and methods and modes (the three forms are called *procedures* in this document) are defined in an object definition.

Defining Operators

Most of the operators can be defined for object types. An operator retains its precedence and its associativity. The operators '.' (selection), AND, OR, NOT, and assignment (=) cannot be defined or redefined.

The binary operators + * - / | & *> <* and ~ can be defined by creating a mode with base type a single argument type, and result type shown, and in place of the function name is the operator *op*:

[*base_type*] **mode** *op*(*type* b) **return** *type* [*var_name*]

where *type* is usually the same in all 3 places. This implicitly defines *op*= as well, when a parameter "(*type* b)" is supplied and *op* is not a comparison. The base *type* can be omitted or it can be **self**.

Equality (=?) is already defined to mean bit-wise equality for objects, but can be redefined for any such types.

If there is no argument "(*type* b)" then *op* can be ++ or - -, defining *op* as a postfix operator, and *op* can then also be + or - or ~, defining these as unary prefix operators.

It is not necessary to define subtraction if both addition and unary minus are defined. Similarly, if =? and > are defined, all other comparison operators can be derived. Addition and multiplication are commutative operators; if two types are used (such as Complex and float) the opposite pair is inferred.

The operators cannot be redefined for the basic types int, float or string.

Defining an operator does not change its precedence or its associativity. Logical operators AND and OR and NOT cannot be defined.

Although operations on basic types cannot be redefined, invalid operations on them can be defined.

A defined operator is not a "true" function; **a + b** cannot be written in function form: **a.+(b)**

The returned value is a reference for a defined object type, a value for a basic type.

Defining Type Casts

Similar to the definition of an operator, a type cast can be defined in an object. The following is the form for defining how to evaluate abc.*type_name*, where abc is an item of type *base_type*:

```
base_type function return type_name [ var_name ]
{
   statements, evaluating a type_name value from the base...
}
```

The base type is the type in the left, the return type is the type on the right. The keyword **self** means the base type of the enclosing object. The word **self** can then be omitted or the type name used instead.

The base type can be an arithmetic type so long as the returned type is a defined type.

Declarations

A *declaration* is a non-executable statement or construct. A declaration defines the type and use of names in the language. The following are declarations:

- A *type declaration* defines variables and arrays and procedures, where allowed, with their type and possible initial values.
- A new onject type definition.
- A **nameset** definition.
- A named constant declaration.
- An **interface** declaration.
- A procedure (**method**, **function**, **mode**) definition.
- A property declaration.
- An *indexer* property declaration.
- An **pragma** declaration.

Declaring Names

Variables and arrays (and new types) can be declared:

- At the beginning of a block, and of a switch case, considered a block.
- In an object declaration.
- A variable with local scope can be declared in a **for** statement initialization clause. It is declared and local if a type is given, but must be known type if not declared.
- Variables for the index and value are implicitly declared, with limited scope, in a **for each** statement. The index is the index type of the array. The value is the same type as the array.
- Variables are declared with type information before them. Types are int, float, string, digital, complex, and any known object type name. Examples:

```
int number
string name
```

- Arrays are declared with square brackets. All members must be the same type and all indexes either uint or int or string or a nameset.
- Variables and arrays can have initial values specified after the name. Examples:

```
int mnx = 23
string ap[] = string['start','stop']      # indexes are 0, 1
ulong salary[string] = ulong['CEO'=250000, 'analyst'=45000,]  # last comma OK
```

- Variable names and procedure names in an object (the same name space) must be distinct. However, a variable can be the same name as a procedure which is defined in a different name space, and vice versa.
- Nameset value names are in separate name spaces for each nameset.

Named Constants

A named constant declaration defines a name which can be used for a constant value. This declaration can appear in an object declaration, the default object, or any block. It has the following form:

[*access*] **const** *type_name name constant-valued-expression*

The *access* keyword (**private** or **public**) is used only in an object type. Access is always local in a block, and follows normal access rules in an object.

The *type-name* is any basic type, or an object type.

The defined *name* is by convention all capitals where letters are used, but this is only a guideline. Predefined named constants begin with an underscore.

The *constant-valued-expression* can consist of a single constant of the declared type or any simple expression using only operators, constants, named constants, object constants, but no properties or functions. It must be convertible to the declared type.

Examples:

```
const int MAX_TIMES 100
const float INTEREST 0.035
const string PROMPT 'User ID? '
const float BETA 4.336e0
const float GAMMA (1 - 1/BETA)
```

Statements

A *statement* is a single line executable action, or a multi-line executable *construct*.

Statements which control the flow of execution, meaning the progression from one statement to another, are called *control statements*. These are the **if**, **for**, and **switch** constructs, and subordinate control statements **return**, **break**, **repeat**, **case**, **default**, **continue**, **else**, and **exit**.

Other statements involve variables and arrays and procedures and expressions. These are:

- The *assignment* statement uses the assignment operators **=** and *op***=**. It can optionally begin with the keyword **set** when it is a single statement on the if or for line as a block.
- The *incrementation* statement is actually a simple expression using the postfix incrementation (**++**) or decrementation (**--**) operator. The side effect change makes this also a statement.
- A *swap* statement, using the swap operator **<>**.
- A *method call* is a statement.
- A *block,* enclosed in curly braces, is considered to be an executable statement.

  A *block* begins with **{** on the end of a **for** or **if** or **final** or **when**, or on a line by itself. It then has a sequence of declarations and statements, ending with a line that has **}** by itself. If a block has just one (or no) statements in it, the curly braces can be on one line surrounding that statement. A pair of braces ("**{}**") is a *empty* statement. It does nothing.

  A block delimits the scope of items declared in it.

  A block can be used anywhere an executable statement can appear.

Control Statements

Single Statement Block

A *single_statement_block* is a single simple (non-construct) statement, treated as a block. The **continue** statement is not allowed . On the if or for statement, the single statement must begin with a keyword. An assignment can begin with the keyword **set** in this case. A single statement block is allowed on the same line in **if**, **else**, **final** and **for** constructs. It is not allowed on **when** blocks.

Any other block starts a new line and the curly braces are on separate lines.

IF Construct

The *if_construct* has the following form:

if *logical_expression block*
[**else** { *if_construct* | *block* }]...

A *logical_expression* is an expression that evaluates to true or false.

An **if** construct is allowed to start after **else**, without enclosing curly braces for a block.

Note that a single_statement_block is a block, and after **else** can be any single statement or a block.

else' is not a statement, it is a clause in an **if** construct.

SWITCH Construct

The *switch construct* provides a set of alternative actions, somewhat like a nested if.

It has the following form:

```
switch expression
case constant | constant_range [,...]
        statements
[ default
        statements
]
end
```

The expression on the **switch** must be an integer or string expression. The constants must all be the same type. The constants can be nameset names. The expression is evaluated exactly once in a switch statement.

Execution proceeds to the **case** or **default** which matches the value of the expression. The following statements are executed. Execution goes to the **end** if it falls through the statements into a **case** or **default**. This is not like the C or Java or similar languages.

The **break** or **continue** statement can be used to override the flow of execution. A **break** sends control to the **end**, and **continue** overrides the implicit jump to the **end** .

The **continue** is allowed only as the last statement before **case** or **default** or **end**. No statements are allowed between **switch** and the first **case** or **default**. There must be as least one case block in the construct.

A *constant_range* is a constant followed by colon then a higher-valued constant. It represents all values between the two shown. If the first constant is omitted, it means all values below the second value. Similarly, if the second constant is omitted it means all higher values. One of the constants must be provided in a range. A range is not allowed for nameset names. The set of constants and constant ranges must not have any overlapping values.

A **case** statement can list multiple values and ranges separated by commas.

The **default** block indicates selection of any other unspecified value. It must follow all **case** statements. There must be at least one **case** block.

Example of a switch:

```
int mv = 8, xlv
switch mv
case 1 : 5
        xlv = 0
case 6 : 25, 50 : 59
        xlv = 1
default
        xlv = -1
end
```

FOR Construct

The *for construct* is the loop statement. It has one of the following forms:

[1] **for** [[*type*] *assignment*] [{**while**|**until**} *test_expr*] [**repeat** *increment*] *block* [*final_block*]

where *assignment* is executed once, first, *test_expr* is a logical expression and *increment* is executed at the end of the block before repeating the statement or block. The **repeat** *increment* and {**while**|**until**} *test_expr* phrases may be in either order. The while or until test always takes place before the block, the repeat part always follows the block implicitly at the start of the final block..

The *assignment* sets a variable to an initial value. If the variable is preceded by a *type*, the variable has that type and has scope local to the for statement and the *block* and *final block*. If untyped, it must be a valid existing typed variable, and the value is retained after the end of the for loop for normal scope.

The *increment* is allowed to be an assignment, an increment expression, or a method call.

Note that the *assignment* or *test_expr* or *increment* portions can be omitted, as for example:

for int k = 1024 **repeat** *k = k/2* **while** k \= 0 *block*

which assigns a value then repeats the *block*, dividing k by 2 at the end. Since *type* is supplied, the variable is local scope in the block. The loop terminates when k is zero.

And,

for {**while**|**until**} *test_expr block* [*final_block*]

which repeats the *block* as long as the test expression remains true (if **while**) or false (if **until**.)

Also,

for *block* [*final_block*]

which repeats the *block* and *final_block* until some condition causes the loop to exit.

[2] **for each** *val* **=** *array block* [*final_block*]

which uses each element of the array, assigning the (local scope) variable *val* the value, repeating the *block*. The type of *val* is the same as the array type. Only the array elements which exist (the index exists) are used. Values are presented in index order, without the index.

If the array has a fixed index, it can be unrolled.

A string can be used as an array. Variable *val* will be assigned each character, left to right,

[3] **for each** *val* **[** *k* **] =** *array block* [*final_block*]

which uses each element of the array, the local scope variable *k* is assigned the index (key) and local scope variable *val* gets the value, repeating the *block*. The type of variable ***k*** is the index type, and ***k*** will begin with 0. The type of ***val*** is the type of the array.

If the array has a fixed index, it can be unrolled.

Example:

```
float prices[string] = float['notebook'=2.55, 'pen'=0.77, 'paper'=1.98]
for each cost[item] = prices
{
   printLn "item={item}, cost={cost}<br>"
}
```

A string can be used as an array. Variable val will get each character, k will begin with 0.

[4] **for each** *obj* **=** *_Traversable_object block* [*final_block*]

which traverses any object *obj* that enables the interface _Traversable, with *obj* an implicitly declared local variable of the same type. This allows traversal of a list of objects for example.

An array is a traversable object.

[5] **for each** *val* **[** *k* **] =** *nameset_indexed_array block* [*final_block*]

yields each value, with **val** implicitly typed and **k** is type **string** for the associated namelist*_name*. The key portion “[k]” can be omitted.

[6] **for each** val **=** *object.generator_function block* [*final_block*]

which repeatedly calls the *generator_function* while val is not null.

FINAL Block

The keyword **final** is used in several contexts - it identifies a finalizer in an object type definition and identifies an object type that cannot be inherited. These uses are not as a statement, but a modifier to the declaration.

Immediately following the *block* on a **for** construct, a **final** statement introduces a *final_block*, the block of statements after **final**. This block of statements remains part of the for block for name scoping. Execution flow falls into this final block, through the **final** statement. The incrementation part of the for statement line is implicitly placed at the start of the final block, if it was specified. In fact, if there is no final block, there is an implicit one containing the incrementation.

The purpose of the *final_block* is to allow multiple actions at the end of each iteration.

The *final_block* is executed on every repetition of the **for**. A **repeat** statement, defined below, transfers to it. A **break** statement skips it,

The format of the combined for and final blocks is:

for *for-conditions*
{
  for_block
}
final
{
  final_block
}

A simple statement block can be used on for and final lines.

REPEAT Statement

The **repeat** statement is valid only inside a **for** block, and is not valid in the final block portion. It causes execution to skip over statements to the incrementation (**repeat**) step for the next iteration of the loop or to the final block. In other words, to the incrementation, the **final** block if it exists, or the next containing "**}**" at the block end on the nearest containing **for**.

BREAK Statement

The **break** statement is used to exit the nearest containing for block or switch or any block. It is typically used to exit a **for** loop when a condition is reached.

It is valid only in a for block, final block, any block which is not a procedure body.

Example:

```
int i   # scope extends after the for block - -
for i = 0 repeat i++ while i < 10
{
   if price[i] < 0 break
   total += price[i]
}
if i<10 { printLn "Error on price number {i}" }
```

CONTINUE Statement

The **continue** statement is allowed only immediately before a **case** or **default** in a **switch**. It cannot be conditional.

It alters the implied break, allowing the flow of execution to fall through into the next case.

EXIT Statement

The exit statement terminates execution of the script. It can appear wherever an executable statement is allowed.

RETURN Statement

The keyword **return** is valid in an object definition. It can be used in a procedure body, in a property declaration, and in any procedure definition or prototype.

In a method, it returns control to the point after the method call.

In a mode, it returns control with a result which is a reference to a base object.

In a method, the return statement has no parameter. A mode can omit the parameter if the base type is returned, or supply a type name. A function always ha a value to return. In any procedure, falling into the end of the definition causes an implicit **return**.

In a function definition, the return statement has the form:

return *expression*

where the expression is the value of the function.

In a function, the last statement in the function must be a return with a value unless there is a return value variable. In that case, a return of that value is implied at the end.

A generator function handles **return** so that the function resumes on each iteration after **return** with a value, or terminates after a **return null** or a valueless **return**.

SIGNAL Statement

This statement causes an immediate transfer to the nearest effective *when_block* to process an error or other condition. It has the form:

signal *condition*

where *condition* is a value of the type **_Condition**, which is a pre-defined object type, in **std.sslib**:

Nameset _ConditionLevel

Used in defining the severity of conditions:

nameset _ConditionLevel N = 0, I = 1, W=2, E = 3, A = 4

with meanings:

N is Normal (no error), I is information only, W is a warning, E is a non-abort error, A is a serious error with an abort recommended.

Object Type _Condition

The object type _Condition has values:

```
object _Condition
{
   _ConditionLevel level      # a nameset
   string message             # a string with a message
}
```

There are standard conditions defined, including, for example these from **std.sslib**:

```
_CON_Success:        _Condition(level=I, message='Success')
_CON_Warning:        _Condition(level=W message='Warning')
_CON_Error:          _Condition(level=E, message='Error')
_CON_Overflow:       _Condition(level=E, message='Overflow')
_CON_DivideByZero    _Condition(level=E, message='Divide by Zero')
_CON_ArrayBounds:    _Condition(level=A, message='ArrayBounds')
_CON_Failure:        _Condition(level=A, message='Failure')
_CON_Null:           _Condition(level=A, message='Null Reference')
```

WHEN Block

A *when_block* can immediately follow a block, a for block (after a final block), a procedure block, or a switch end statement, or a block not part of if or else, but not a case block, and not prior to a final block.

It has the keyword **when**, followed by a list of allowed conditions, followed by a block. This block is executed only when a **signal** statement has been executed. Otherwise, it is skipped.

The *when_block* shares the name scope with its prior block.

Upon entering the *when_block*, the predefined variable **_ConditionPosted** is implicitly defined, of type **_Condition**, with the value of the signaled condition.

If the block does not want to process this condition, it can pass the condition to the next outer *when_block* by executing the statement:

```
signal _ConditionPosted
```

If the global area is reached, a system-supplied action is performed for that condition.

A condition can be ignored:

```
if _ConditionPosted.level =? I break # ignore information level
```

Blocks

A block, with curly braces, acts an executable statement. The block can contain declarations and statements, intermixed. The statements and declarations are performed in order, from the top, unless a control statement changes the order of execution.

The names defined in a block have scope from the definition to the end of the block. Access modifier **public** is not permitted. If **private** access is specified, the scope does not extend to inner blocks.

There are also other implicit blocks, in the sense that they are groups of statements and declarations, and the declarations have scope like a block delimited by curly braces. These blocks are:

- The *object block*, all statements and declarations in an object definition. The scope of these names is special. Variables and arrays are protected, and types defined are private. Procedures and items are private if that keyword is used.
- The *for block* includes the entire **for** statement and the block executed. It also includes the *final block*.
- The *case block* is all of the statements and declarations from case or default in a switch until case or default or the end of the switch.
- The *switch block* is all of a switch construct. It includes the expression on the switch statement plus all of the enclosed case or default blocks.
- The *procedure block* is all statements and declarations in a procedure definition. This begins at the start of the prototype portion and ends at the ending curly brace. This applies also for operator and type cast definition and for a constructor or finalizer. It also applies to variants.
- The *final_block* is the statements enclosed in braces after **final,** in the following block. It is not a true block because it is part of the same scope as the for block.
- The *when_block* is the statements from **when** through the following block. Its scope extends the preceding block.

Objects

A *object* in SS is a named entity which has values, procedures and other properties within it. Objects are used to encapsulate data and properties, hiding details.

The object name becomes a type, an *object_type*, which is used to declare variables and arrays and procedures, etc. An *object_type* is analogous to a *class* in many other languages.

An object can inherit another, gaining its properties. Multiple inheritances are limited.

Generally, a new *object_type* is defined in the following manner:

```
[options] object object_type [ ( param_type_spec ,... ) ] \
          [inherits object_type [ ( variant_name, ... ) ] ,... ] \
          [enables interface_name ,...]
{
    [ variable and array declarations ] ...
    [ property declarations ] ...
    [ indexer declaration ]
    [ procedure definitions ] ...
    [ abstract procedure definitions ] ...
    [ constructor definition ]
    [ finalizer definition ]
}
```

where *options* is any of the keywords **final**, **abstract**, **set** or **default**.

If the options keyword **final** is used, the object type cannot be inherited. It can be used to type data or procedure items. The object type must not have any abstract procedures in this case.

Options keyword **set** forces any parameter passed to a procedure, not the base item and not an explicit **ref** parameter and not a parameter of a defined operator, to evaluate a copy to be passed. It still passes by reference.

If a parenthesized *param_type_spec* appears, it represents parameter names, for example:

```
(ParName: int float)
```

where the parameter ParName is a type, such as **int** or **float**, or an interface name. Additional param_types may appear after a comma, A type preceded by a reverse slash is not allowed, after a specified interface name. One type may have an equal sign, indicating a default type.

An *enables_spec* is an *interface_name* list (separated by commas) appearing after **enables**. It may appear before or after an **inherits** specification. The **enables** and **inherits** phrases may be in either order.

Variant names listed must exist.

The inherited *object_type* can also be one of the basic types (int, uint, logical, string, digital or complex). This becomes an *enhanced_type* object definition.

Object definitions can be nested.

Multiple inheritance follows the rule: the order of specification in the object declaration affects the name resolution for members using the C3 linearization algorithm

Variable and Array Declarations

Variables and arrays are defined in the form

[*access*] *type_name* name [array_spec] [**=** initial_value [**final**]] ,...

Variables and arrays in an object are visible only within the object, or (if not **private** access) in an inheriting object (like 'protected' in C++) .

The keyword **final** following an initial value specifies that the value is immutable, or read-only. It cannot be set to a new value. This applies also to static values.

A variable definition example:

float rate = 0.035

Static Values

A variable or array in an object can have the keyword **static** preceding the type. This implies that there is a single shared copy in the program for all objects of this type. It is not in an instance of the object, but belongs to the object type.

These static items are not accessible as members of a variable. Access is through the object type name as a base. For example, static value **val** in object type **ABC** is referenced as:

ABC.val

Static values do not go away when objects are deleted or out of scope.

Static Procedures

A method or function may have the keyword static, meaning it is invoked only by referencing the type name as a base. It may not use **this** since there is no instance. It may only use static variables.

Static is not allowed on a mode.

Property Declarations

A property is a public name acting as a variable with controlled access, allowing read-only or write-only or requiring computation or procedure calls.

The format is:

[**abstract**] *type property_name* [*set_phrase_list*] [*return_phrase*] [**,** …]

where:

- The word **abstract**, if present, means the property name must be defined in an inheriting or enabling object type. This is an *abstract_property*. The set and return keywords must be without action methods or expressions. A property declaration in an interface is inherently an abstract property; the word **abstract** is optional in an interface.
- In an object definition, the object must have the abstract keyword if it contains an abstract property.
- There is no access specified. It is public.
- There is no initial value.
- *property_name* is a variable name. It acts like a function argument in the set phrases, and is not used in the return phrase.
- *set_phrase_list* is one or more of these:

  set { *internal_item* = *expression* | *method_call* }

  where at least one usage of the *property_name* is used.
- Multiple set phrases are executed left to right.
- r*eturn_phrase* is: **return** *expression.* If the expression is a constant value, and there is no *set_phrase*, the name is an alternate form for a public named constant.
- The set and return can be in either order, and there can be multiple set phrases. When there are multiple set phrases there is no return phrase intermixed.
- There must be at least one **set** or **return** phrase**.**
- A property item can affect any accessible internal values.
- In an *interface* or when the **abstract** keyword is used, the expression for **set** or action for **return** must be omitted. It is supplied when the definition is made later.
- Property definitions can be intermixed with variable definitions which do not use public, private or final.

Note that a property value can be read-only (it has no **set**) or write-only, with no **return** keyword. A set phrase can also be a method call.

A *property_name* may or may not allocate space in the object. It cannot be static. It must be a variable.

The property names are referenced like members of an object. They are always public.

Example Of Property Usage

An example of property usage, allowing conversion between inches and centimeters:

```
object  Dimens
{
  float wid, len # can be set only inside the object (internal values are inches)
  public string item_name
  float width return wid * 2.54, set wid = width / 2.54 # outside is cm
  float length return len * 2.54, set len = length / 2.54 # outside is cm
  float area return wid * len / (2.54*2.54) # no set, returns square cm
}

Dimens door
door.item_name = 'Rear entry'
door.width = 125
door.length = 300
printLn "{door.item_name} is {door.area} square cm.<br>"
```

Indexers

An *indexer* is a special form of a property. It provides a way to define the access of array elements of any array of the object type in which the indexer is defined.

The form of an indexer is:

[**abstract**] [type] **[** *index_type index_name* [= *max_index_name*] **]**
[*return_phrase*] [set_phrase]

The keyword abstract has the same meaning as a property declaration.

The *index_type* and *index_name* are used like a subscript. Index type is a basic type suitable for an array index. A fixed index is indicated by a *max_index_name* after the index name, following an equals sign. A nameset index iis shown by the index_type as the word **nameset** followed by ain identifier to use to refer to the nameset for properties.

The return phrase must be present except in an abstract indexer or an interface. The return phrase returns a reference to the indexed member of the object.

There can be several indexers in each object type, one for each index type. An abstract indexer can be in an interface, similar rules to any property.

A typical use of an indexer is to define the array as a tree or list. It also may define rules for a fixed size array using buffering.

Object Constants

Object members are by default protected, so only the property values which have a set phrase or variables or arrays which are public may be used in an object constant. Inside the object definition or when it has been inherited, the values may also be accessible non-public names.

The object type name is used, followed by a set of name=value assignments in parentheses. All value assignments are optional and may appear in any order. Default values are supplied as needed.

Example:

```
object Animal
{
   private string k
   string kind set k = kind return k
   public string says
}

object cat inherits Animal = Animal(kind='cat',says='Meow')
```

Inheritance

When an object type inherits another, it gains access to the public and protected variables and procedure definitions.

An object type definition can name a single parent object type to be inherited.

A variable *V* of a given object type *T* fits the words: *V* "is a" *T*.

If *T* inherits *TT*, then *V* "is a" *TT* also.

As an example, with V1 of type CAR and V2 of type SUV, and each of these types inherit a type AUTOMOBILE, then we can say both V1 is type AUTOMOBILE and V2 is type AUTOMOBILE. This allows a procedure to process either V1 or V2 as an AUTOMOBILE.

Every object (all things are considered "objects") supports this built-in function to test or inquire about the object's type:

```
isType          - - property, returns the type name as a string.
```

The type name is returned in lower case for builtin types.

Identifying the Base Object

Outside an object type definition, an object name is used to identify the base object, followed by a period as an object member selector, then the member or property name or the function, method or mode as appropriate.

Inside an object type definition, the current object is assumed for any name defined in the object or inherited. The base name can be **this** or a visible object's name.

If the name overrides an inherited object type and the intent is to refer to the parent object's variable or procedure name which is obscured by the override, a base name is **parent**.

Visibility of Names - Access

The keywords **private** and **public** control the places where a name of a variable or procedure or an object type are valid, or visible. This is the *access* specification.

When none of these are specified, the item is “protected” and is visible across the object and any inheriting objects.

The keyword **private** restricts visibility to the object, no inheritors.

To make items public, the keyword **public** is used.

Enables Specification

The *enables_spec* appears after the *object_name*. It states that the object type implements the interface names listed.

The *enables_spec* has the format:

enables *interface_name,...*

It indicates the object defines the features named in the interface. An object which inherits another which enables an interface also enables that interface and must not violate the specification.

Instantiation of an Object

An object variable is a place-holder for a reference to an object. It is instantiated when it is assigned a reference to an instantiated object value. Creating an object constant using the object type creates a value and instantiates the object. Assigning or initializing a value inside the object also instantiates it. The static components are instantiated when the object is declared.

Default Value at Instantiation

An integer, float variable has a default value of 0. Complex default is complex(0,0).

A string has a 0-length string value as default.

A logical variable defaults to **false**.

Object Member References

A property name can be referenced as a public name. Internal items are not public. The object name is followed by a period, then by the property name or the procedure name.

The property name reference can appear in an expression, returning a value, only if there is a return phrase. It can appear as an L-value (a left side in an assignment) only if there is a set phrase.

It can be the base of a further reference only if it has both set and return phrases. It can be an actual argument only where the set and return phrases support the usage of the matching formal argument.

It is better to use an object as the base of a procedure than to pass it as an actual argument.

Type Switch Construct

The *type_switch* construct provides a set of alternative declarations, only in an object definition. It has the following form:

switch *param_type_name*

case *type name* | interface_name [,...]

[*declarations*]

[**default**

[*declarations*]

]

end

This can only be used in an object definition which uses a *param_type_name*.

The type names on each **case** must be unique, There must be at least one **case**. A type name can be used on more than one **case**. A **continue** or **break** is not allowed.

No executable statements are allowed in this switch except within declared procedures or properties. This kind of switch is not allowed inside a block or procedure.

This construct allows embedded procedures to be customized for the *param_type*.

Enhanced Object Types

The inherited object_type can be one of the basic types (int, uint, logical, string, complex or decimal).This becomes an *enhanced_type* object definition.

The enhanced object is restricted in several ways:

- It can contain only one non-static data member, of the same type as is inherited.
- It can contain static members, also constants.
- It can contain properties and procedures which do not use local values.

Like any inherited type, the new type is a type that is inherited. For example, a new type Degrees which inherits float is also a float.

The enhanced object type is a way to specialize values, One enhanced type cannot be mixed with another even though they both inherit the same type. For example, a Meters type and an Inches type can inherit float, but they cannot be mixed by mistake. Properties may be provided for conversion. Automatic type casting does not happen.

An enhanced type is permitted to be used as a value of the inherited type. This allows mathematical functions to be applied. However, if an expression mixes types, the mix is restricted. Addition and subtraction must use the same type, and the result carries the type, so inches plus inches is still inches.

However, multiplication and division is allowed only with the inherited type. This allows “scaling” or “discounting” to happen.

An enhanced type cannot be cast as its inherited type, losing the enhancement. This protects from switching “units” like trying to typecast a metric Length to inches by using **Length.float.Inches**. The correct way is to provide properties or procedures which convert directly.

An object which inherits an enhanced object type is not restricted. It may contain other data.

Defining Procedures

Procedures can be defined inside an object definition. Procedure definitions cannot be nested. They can also be based on a standard type.

A procedure can be defined on any array of any type by specifying an object type of **[]** - a pair of square brackets. If it is preceded by a parameter type specification, with or without the array indication, that can restrict or specify types the procedure applies to. The array index type can be specified to apply to a fixed or a specific type of index.

For all procedure definitions, the *base_type* can be omitted if it is **self** or the containing object.

On all procedure definitions, *access* is one of the keywords **private** or **public**. If none is specified, the access is the default, "protected."

Method

A *method is a* procedure with no return value. No return specification is provided. The general form for a method definition is:

[**abstract** | **static**] [*access*] *base_type* **method** *name* [*args_spec*]
{
    [statements and declarations] …
}

Function

A *function* always returns a value. The general form for a function definition is:

[**abstract** | **static**] [*access*] *base_type* **function** *name* [**(** *args_spec* **)**] [**repeat**] *return-spec*
{
    [statements and declarations] …
}

where the option keyword **repeat** implies this is a *generator_function*. The *args_spec* is omitted if there are no arguments. The *return_spec* specifies the returned type and an optional local variable name after the keyword **return**. It is required. The return type must not be a defined object type; a **mode** is required for that usage.

Mode

A *mode* is a procedure with or without arguments, based on a defined object type, which implicitly returns a reference to the object. It may return instead a reference to another *object_type*. The default return type is **self.**

A mode is assumed to set (modify/alter) internal values in the base object, or create a new object. The general form for a mode definition is:

[**abstract**] [*access*] [*base_type*] **mode** *name* [**(** *args_spec* **)**] [*return-spec*]
{
    [statements and declarations] …
}

Non-Object Procedures

A non-object procedure is not based on an object type. The base_type can be a basic type (**int**, **float**, etc.). Methods and functions can be defined, modes cannot. A method can be defined on **null**, but a function or mode cannot be on **null**.

A non-object procedure must be defined in an object or default object definition. It cannot have a modifier **private** or **public** or **abstract** or **static**.

If a method is defined on **null**, the invocation has no base.

Procedure Syntax

The base_type is omitted or it is the keyword **self** in a procedure defined in an object. If omitted, it is understood to be the object's type.

A base_type of **null** can be used to define a non-object method.

The keyword **private** is allowed on a procedure definition inside an object definition or interface. It means it will not be inherited.

Procedures in an object are defined with any appropriate access.

The keyword **abstract** on a procedure definition inside an object definition or interface means there is no procedure body given. The procedure must be defined by an inheriting object. The abstract procedure is a template.

The Base Type

The *base_type* is omitted or is **self** when it is the containing object type. Specifying **parent** implies it is based on the inherited type.

In a default object definition, the *base_type* on a procedure definition is omitted**.** This means the *non-object* function or method (it cannot be a mode) has no base type and no base argument.

A procedure with a *base_type* is an *instance procedure*. It must be invoked on an object of that type. The *base_type* can be a defined type or a basic type.

The *base_type* can specify an interface parameter type mentioned on an enabled interface definition.

The base can be declared as any object enabling (implementing) an interface name. In that case, the base*_type* is an interface name and the base object must be an object which enables the named interface.

The Arguments Specification

The *args_spec* defines the expected arguments for the procedure.

The *args_spec* is omitted for no formal arguments, or it is:

{ *arg_type* | *interface_name* } [****] [**ref**] *arg_name* [**=** *constant_expression*] [, …]

when there are arguments. The arguments are separated by commas.

If the keyword **ref** appears before an argument name, it indicates the variable is passed by reference rather than by copy and that the value can be changed in the procedure. An array or string or flag or named object type by default is passed by reference. A reference item can use **ref** on an argument to force a reference item to pass a reference to a copy.

A reverse slash before an argument means it is not allowed to be modified. This is default behavior for defined operators.

Argument Passing

All value variables (types int, float, logical) are passed by copy, unless the keyword **ref** is used.

A variable passed by copy delivers a temporary copy to the procedure. If the procedure alters the copy, the original is preserved. To pass a copy of any object, append it with the built-in function “copy” after the object name.

An expression always passes a copy. An item enclosed in parentheses is an expression.

An object type implies the actual argument is an object of that type or it inherits that type.

An argument can be declared as any object enabling (implementing) an interface name. In that case, the *arg_type* is replaced by an interface name and the actual argument must be an object which enables the named interface.

Arguments which are variables may have a default value expressed as a constant expression. Any such argument can be omitted in a reference if it has no following explicit actual argument. The value passed is the value of the constant expression.

An argument can be a procedure argument, specified by a procedure prototype. The actual argument will be a matching procedure name with its base type shown.

The Return Specification

The *return_spec* is required for a function, not allowed for a method, optional for a mode. It has the format:

return *result_type* [*return_item* [= *init_return_value*]]

If the *return_item* is supplied, it is an internal variable or array name which will be returned. The return statement can omit the value in this case. A return is implied at the end of a function. The *init_return_value* is an initial value expression for the return variable.

The *result_type* or *arg_type* or *base_type* can be a basic type such as **int**, **float**, **logical**, **string**, **digital** or **complex**, or it can be an object type name, and it can be an array specification (with type and index type), an interface name or an interface parameter type mentioned on the containing object definition.

When *base_type* is an object type or any defined type, the procedure has access to internal data inside the definition.

The base object is implicitly passed by reference, regardless of type.

If the *result_type* can be an object or an array, or if it is a generator function, a function is allowed to return the constant **null** as an indicator of failure, meaning it failed to create the object.

The Procedure Body

The embedded statements in the procedure (the *body*) define the actions and internal variables used. In place of these statements, the reserved word **pragma** can be used. It has parameters which indicate the implementation is defined by another language or by externally provided code, or code known to the compiler, or it may indicate conditions governing the compilation, such as inline code generation.

In an interface declaration, the *body* is omitted, which leaves only the prototype.

Within any procedure defined in an object, the following keywords have defined meaning:

this means the current instantiated object. It can be used in an expression.

self means the current type.

parent means the base type of the inherited object type.

Abstract Procedures

If the *body* is omitted on a procedure in an object definition, the procedure is an abstract procedure prototype. The keyword **abstract** is then required before the prototype except in an interface. An inherited abstract procedure must be defined in the inheriting object type. The abstract procedure remains abstract in inheriting objects if not defined, carrying the required definition to the next level.

Variant Procedures

The procedure definition in an object can offer variations in the definition block when the procedure is not abstract. The first definition block is the default. A variant can be chosen with the **using** phrase when inheritance is specified. A property cannot have a variant.

Each variant is defined after the procedure block with a variant declaration:

```
variant variant_name [return_spec]
{
   procedure body
}
```

Variants provide a way to avoid a need for abstract procedures and avoid a need to define additional object types. The feature is a more controlled and protected subset of “traits” as in the PHP language. It is not a means for inclusion.

The variant assumes the same *args_spec* as the procedure defined.

The *return_spec* is permitted on a mode or function variant. If omitted, the return is the same as the base mode or function. If provided, it must be different from the base mode or function.

A *variant_name* procedure is not referenced by name anywhere in an object definition. It is chosen with the **using** phrase when an inheriting object is defined.

Constructors

A *constructor* is optionally declared in an object type definition. The constructor is called when the object is instantiated, or when the object is initialized. A constructor appears in the object type definition and has this format:

```
[static] default mode [args_spec, ...]
{
   declarations and statements ...
}
```

There is no *base_type* and the implicit mode name *type_name* is the name of the object type when the constructor is invoked. The constructor can be invoked as a mode using the type_name as a way to create a new object.

The can be at most one static constructor. It is used only for the first instantiation and it can only define static values.

If there are no arguments on a non-static constructor it will be invoked by default when the object is instantiated. If a constructor is not defined, default instantiation occurs. The constructor does not have to be defined.

A non-static constructor is invoked by the type name and it provides a new instantiation.

The statements can use temporary variables, etc., and they can access the items in the object.

One purpose for a constructor is to set static values and private members and perhaps open a database.

A constructor must have statements in the block. The block cannot be empty.

There can be multiple constructors for different arguments providing different types of values to be converted.

Finalizer

The *finalizer* is also optional, and is declared in an object type definition. The finalizer is called once when the instantiated object is finally destroyed. It has **final** before the keyword **method**:

```
final method
{
   declarations and statements  ...
}
```

The statements can use temporary variables, etc. and they can access the items in the object.

One purpose for a finalizer is to clear values, clear out buffers, and perhaps close a file. It is the object destructor.

The finalizer (if defined) is called when the object goes out of scope or the last instance disappears. It also is called if the built-in method **Clear** is applied to the object. An object can have at most one constructor and at most one finalizer.

A finalizer must have statements in the block. The block cannot be empty.

Generator Functions

A *generator_function* is defined with the keyword **repeat** before the *return_specification*.

It can be used only in a **for each** statement.

Each time it returns a value, the current state is preserved, and when invoked again, it resumes where it left off. It finally quits, indicating an end to the set of values, when it encounters either a plain valueless **return**, or **return null**, or the end of the function body.

A generator mode is similar.

Invocation of a Procedure

A procedure defined on a given type is invoked by naming an object of that type, then the period (selector character), then the procedure name with arguments as needed.

Invocation of a method will not use parentheses.

A mode or function invocation does not use a pair of parentheses when it has no arguments. A sequence of applied modes is called a *mode stack*.

If there is no base type (it is defined in a default object or on **null**), the base and the period are omitted.

When a function or mode returns a reference to an object, that object may have another procedure or property variables. A function or mode invocation can then be followed by a selector period and another reference. Invocations can be nested or stacked.

When an expression of any kind is passed, a copy is passed rather than a reference to the original value. This applies even for a parenthesized item, or for type casts, or for a unary plus. In other words, **(abc)** is a copy of **abc**, as is **+abc**.

The *base* object (on xyz.MethodCall for example, the base is xyz) acts as if it is passed as an argument named **this**, which is passed with an implicit **ref**. When the base is a constant or or expression in parentheses a reference to a copy is passed.

A constructor can be explicitly invoked by the type name. It is a mode, returning a new instance or initializing the current new instance.. It always uses parentheses. For example, the parent constructor is invoked as **parent()** or where the parent type is known to be Defg, as Defg().

A property has no arguments.

The Default Object

When an object definition has the keyword **default** and no object name, special rules are in effect. Such an object type is called the *default object*. The default object does not define a new type. It cannot define an instance. It has no static variables. It actually is not a true object at all.

The default object can be defined only outside any other object type.

Each default object definition implicitly "inherits" previous default objects, and cannot inherit any other object, and cannot enable any interface or abstract items.

It can have property variables.

The default object cannot have a constructor or finalizer. It cannot internally define any object type.

Named constants can have global scope by defining them in the default object. For example, the mathematics constants pi (π) and ***e*** and others are defined as float values to a large number of digits in the default object by these lines (shown truncated) in **math.sslib**:

```
const float _MATH_PI 3.14159265358979323846264338327795 # ...
const float _MATH_E 2.71828182845904523536028747135266 # ...
const float _MATH_LN_10 2.30258509299404568401799145468 # ...
const float _MATH_ONE_DIV_LN_10 0.43429448190325182765 # ...
```

Because the default object can be extended (it inherits prior versions), additional named constants (etc.) can be added. It is recommended that the names of constants should be all upper case.

The names defined in a default object obey normal scope rules. They are defined as global names, as if the default object is inherited by every other object.

Methods defined inside the default object which are not defined on a type have no base type. They are defined on **null**. This means the method is not based upon any object. They are not invoked on a base. . They do not complete the definition of an abstract procedure.

Modes and functions cannot be defined on the default object.

Procedures in the default object are allowed to access previously defined constants, and they are allowed to use global-area type definitions.

Built-in Object Procedures

Table: Built-in Object Procedures and Properties

Procedure	Kind	Result
Clear	method	the finalizer method is called on a named object
copy	mode	a copy of the item
Equals(object b)	function	**true** if the types are the same and values are equal
indexType	property	String - the type name of the array's index
sNotNull	property	**true** if the ref to the object is NOT **null**
isNull	property	**true** if the ref to the object is **null**
isType	property	String - name of the type of the object

The above functions and properties can be applied to any object, as well as a variable or array which is treated as an object.

Name References in an Object

Names within an object or inherited in an object are referenced without need to imply they refer to the current instantiation or static item. If a name overrides the same name, it is possible to refer to the parent's version by affixing it with a selector period and the object name inherited or the word **parent**.

Example:

```
object ABC
{
   string x = 'Hello'
}

object DEF inherits ABC
{
   string x = 'World'
   default mode
  {
      printLn "{parent.x}, {x}!"
   }
}

# use it -
DEF greet  # prints: “Hello, World!” on instantiation
```

Within an object, a reference to the whole object is written as the keyword **this**.

Interfaces

An *interface* is a declaration which defines the prototypes of a list of object functions, methods, modes, type casts, indexers and property values. The *prototype* of a procedure is the initial declaration without the body part (the implementation part.)

An interface is used to enforce or manage the capabilities of an object type, and to define the type of objects which a function or method or mode operates on.

An interface is not allowed inside an object definition.

An interface declaration has this format:

interface *interface_name* [**(** *param_type_spec, ...* **)**] [**enables** *interface_list*]
{
  [*procedure_prototype*] …
  [*property_decl*] …
  [*type_cast_decl*] …
 [*indexer_decl*]
}

If a parenthesized *param_type_spec* appears, it represents a parameter name, to be supplied later with a blank-separated *type_name* list, when the interface is used, as for example:

(ParName: int float)

where the parameter ParName is of type **int** or **float**. Additional param_types may appear after a comma. An *interface_name* can be a *param_type*.

A *type_name* which is to be excluded from an interface which has been named follows the interface name with a reverse slash preceding. For example, the following *param_type_spec* allows any comparable type except **float**:

(key_type: _Comparable \ float)

One type may have an equal sign preceding, indicating default type.

This provides a way to parameterize interfaces. For example, interfaces can describe a "tree of objects of a type." The *param_type_spec* and its parentheses can be omitted.

A *property_decl* begins with the *type*, or a *param_type*, followed by a name, then the words **set** and/or **return**, without specifying the actions, as applicable. Either set or return or both must be present. The enabling object must define those specified in the interface.

The property, with a return type only in the interface can match (in the object) a public variable, a fully defined property, or a function or mode with the same description.

An object type can enable (implement) multiple interfaces. It can then be used as an argument or base for any procedure that requires any of the interfaces in that argument.

Unlike an object declaration, there are no variables defined in an interface. Properties are allowed.

The matching items in the object need not appear in the same order as in the interface.

An interface that enables another interface inherits that interface definition, but it can overload

any part of the enabled interface.

An object that enables an interface must complete the implementation of all items in the interface and those inherited by the interface.

Instead of completing the implementation, an abstract object can define a procedure as **abstract**.

Example of an interface:

```
# a double linked list
interface _TwoWayTraversable enables _Traversable
{
   # inherits Next from _Traversable
   self Prev set return # backward link
}
```

The above interface implies that any object that enables _TwoWayTraversable also enables _Traversable. An **object** declaration which enables _TwoWayTraversable need not say explicitly that it also enables _Traversable.

An object that enables a parameterized interface must identify which actual type the object definition satisfies, or it may instead use a type wherever the parameter name appears in the interface, and thus enables that type.

For example, the following interface:

```
interface Smaller (num_type: int uint float)
{
    num_type function Reduce return num_type
}
```

can be enabled (implemented) by an object that uses integers:

```
Ob object enables Smaller(num_type: int)
{
   int function Reduce return int
  {
      return this - 1
   }
}
```

If the object supports any type, the actual type list is an asterisk.

Interfaces and Procedures

The *base_type* of a procedure in an object which enables an interface can clarify which interface parameter types are enabled by naming the object type optionally followed by parentheses enclosing the parameter types supported. This appears after the object type, and has the form:

(*param_type_name***:** *actual_type_list* , ...**)**

where *actual_type_list* is a blank-separated list of known types, or interface names or an asterisk, meaning "any type."

The order of the type names is not important.

Standard Interfaces

Special interface definitions are supplied in standard library **std.sslib**. They all begin with an underscore character and an upper case letter, as do standard predefined object types.

_Equality Interface

The interface **_Equality** is defined as:

```
interface _ Equality (data_type: *)
{
  function =?(data_type d) return logical
}
```

This interface is implicitly implemented by all objects but can be overridden.

_Comparable Interface

The interface **_Comparable** is defined as:

```
interface _Comparable (data_type: *) implements _Equality
{
  function >(data_type d) return logical
}
```

This is a very simple interface, which implies a means for sorting or comparison. Note that it compares an object to a data item, not necessarily the same type. The intended use is an object which is a wrapper for data, comparing that data with a value. The value may be in another object.

The arithmetic basic types and string type all implicitly enable _Comparable.

_Traversable Interface

The interface **_Traversable** is a very simple interface, which implies a means for selecting a list member. It is defined as:

```
interface Traversable (key_type: _Comparable)
{
   key_type First return
   key_type Next return # supply a 'next item' on the object
}
```

When there is no traversable Next member, the Next return value is **null**.

Any object *obj* which supports _Traversable can be used in a **for each** statement. The **for each** statement internally expands to initialize with the object, then increments (or iterates) using Next, and is stopped when **null** is returned. Thus, the statement:

for each item **=** obj *block*

internally expands into:

```
for item = obj.First while item.Next.isNotNull repeat item .= Next
{
   block
}
```

To make this statement go through a list, for example, the initial obj is the first item of the list. The variable item is a name of local scope, same type as obj.

_Sortable Interface

This interface implies _Traversable plus the ability to swap data values in place. It is defined as:

```
jnterface _Sortable (data_type: *) implements _Traversable
{
   function <>(data_type d) return logical
}
```

Basic type arrays are sortable.

Pragma Declaration

The **pragma** reserved keyword introduces a declaration which informs the compiler about how to compile code. Following the keyword **pragma** are special options and code generation instructions. Some of these may be restricted only to source from standard directories or not available except in standard code.

Some (preliminary) forms which are (expected to be ??) implemented:

```
pragma external 'code file source' # using this as a procedure body allows any code
pragma inline # makes this procedure expand inline, no procedure call used
pragma unroll # instructs compiler to unroll the following for loop if possible
pragma if ... # makes an if construct (one line) conditionally compiled
pragma include … # bring in precompiled code - NOT source code
```

Certain behavior defaults for generated code are controlled by pragma:

```
pragma option
```

Member Access

Access is the degree of encapsulation, or the accessibility of names in an object.

Access for names defined in an object depends on a number of keywords on the object line or on individual declarations, or there is a default access.

Access in SS objects differs from access in classes for Java, C++ and C# in some ways.

The allowed declarations in an object are:

- nameset
- type declarations for internal variables and arrays
- property items
- procedure definitions - functions, methods, and modes
- internal object definitions
- **pragma** declarations
- constructor and finalizer definitions - not visible outside the object

All internal variables, arrays are by default visible (accessible) in inheriting objects . They are not public. This default behavior is like the "protected" access in the languages Java, C++ and C#, which have a different default.

The keyword **private** on a data member declaration limits the visibility to the object.

Internal object types and defined types declared in an object are public unless they have the keyword **private** as an option.

Property items are variable in appearance, but may in fact be implemented as a procedure, or they may be implemented as a public variable or array. They are possibly limited to read-only or write-only access.

Keywords **final**, **static**, **private** and **public** on the object definition affect access.

Printing Strings

The standard library member **std.sslib** defines some built-in (default) procedures. One method which is defined is named **print**. This method takes a single argument, a string, and writes it to the standard output stream or back to the connected browser or other connection in effect. The method **printLn** adds end-of-line characters to the end.

Example:

```
string helper = 'Watson'
printLn "Come help me, {helper}, I need you!"
```

Note the string insertion in the message.

The method names print and printLn are not reserved words. They are in the default object as single argument methods.

Method Named 'die'

Another useful standard method is **die**, which has a single argument, a string. It prints the string, then does an **exit** statement, terminating execution.

Hello, World!

The standard "Hello, World!" program in SS is very simple:

```
null method _Main
{
   printLn "Hello, World!"
}
```

HTML Form Data

Values sent by an HTML form or by parameters attached to the URL are obtained through a single mechanism. The standard library member **std.sslib** defines an object **_FormData** which is used to get these values. Instantiation (the constructor) fills the values in the object member **fields**, an array of strings with string indexes, defined as:

public string[string] fields

This is filled with the passed values when the _FormData is instantiated. The constructor has one argument - the value 'GET' or 'POST'. Default is POST. The string keys are the data names with the values as passed. Password encryption is already decrypted by the server.

The program can easily get the values passed. For example, a form passes a value 'name':

```
_FormData form # instantiation fills in the data
string name = form.fields['name'] # variable name now has the value.
```

Cookies

The standard library member **std.sslib** also defines a way to access cookies, with object type **_CookieData**. This object defines several data fields and methods, The data fields are:

name, value, expire, path, domain - all are type string

each must be an appropriate value. (Details TBD)

There is also a method in the object, write, with no parameters. This must only be used before any output is created.

HTML Headers

The same library member **std.sslib** also defines a method **sendHeader** with a single string argument. The method is not based on an object.

Environment Strings

The same library member **std.sslib** also defines a function, not based on an object, named **getEnv** with one string argument which returns the string value of the Environment string named in the argument.

Directories

Handling directories (aka folders) is done through an object type **_Directory**. This object type defines some functions and methods to create and manipulate directories. The standard library member **directory.sslib** defines the object type and procedures.

To work with directories, a program must first create the object using the constructor, which has one optional argument **path** with a default value of **'.'**, the current directory.

Table: Built-in Directory Functions and Properties

Function:	Result Type:	Description:
changeDir(string path)	_Directory	Change to directory or path (a mode)
clearDir(string path)	logical	Deletes the directory and all of its contents
createDir(string path)	logical	Creates a new directory or subdirectory
currentDir	string	Property, returns the current directory name
dirs	string array	Returns the directories in current directory
files	string array	Returns the files in the current directory
files(string path)	string array	Returns the files in the named directory path
findDir(string path)	logical	Tests if a path is a directory
getPath(string path)	string	Returns full path name of a directory path
renameDir(string name)	logical	Rename/move the current directory

All of the above are based on an object of type _Directory.

Failure when the result is not type logical is a return of **null**.

The meaning of *path* is system-dependent. In the above, *path* can be a single directory or a full path name or a relative path. For consistency, a forward slas is always used.

The function dirs returns the subdirectories, without the pseudo-directories ".." or “.”.

These names can be obtained:

```
_Directory thisDir
string curDirs[] = thisDir.dirs
```

To change to the parent directory, do this:

```
thisDir.changeDir('..')
```

To see files in the current directory as an array of the file names, no directories included:

```
string myFiles[] = thisDir.files
```

It is possible to work with files or directories which are not in the current directory.

Additional functions for security settings are provided.

The File System

SS treats files as objects of a type named _File rather than using file handles or file pointers as in older languages based on C. The definitions are in standard library member **files.sslib**.

The attributes of a file are defined in object _FileAttribs, also in the same library member. This object is inherited by a _File.

From a directory, we can get file names in that directory. A simple name is in the current directory, and a path name refers to any file.

Table: Built-in File System Functions, Methods and Properties

Function/Property:	Base:	Result Type:	Description:
attribs	_File	_FileAttribs	Get the attributes of a file
delete	_File	(none)	Removes a file
extension	_File	string	Returns the extension
file	string	_File	_File object for a path or name
isFile	string	logical	Tests if a path is an existing file
isLegitName	string	logical	Tests if a path or file is a legal name
isWriteOK	string	logical	Tests if a file is writable
name	_File	string	The name part, with extension
path	_File	string	The full path for the file
rename(string name)	_File	_File or **null**	Change the file name
move(string path)	_File	_File or **null**	Move the file and change its name

These _File objects are not prepared ("opened") for input/output. They simply define the files known to the program.

Table: _FileAttribs Object Properties

Exposed Name:	Set/Return:	Result Type:	Description:
owner_uid	return	string	Owner's user ID
group_id	return	int	Group ID (UNIX? LINUX?)
Node	return	int	The file's Node (UNIX? LINUX?)
permissions	return/set	(??)	Permission bits (UNIX? LINUX?)
size	return	ulong	Length in bytes
file_type	return	string	Type

FUTURE Permission bits - make it set or return by *owner*, *group*, or *other*, to *read*, *write*, *execute*. Allow also a form of octal in a string?

File Operations

File operations means manipulation of the contents of files. In order to read or write, a file must be opened. Unlike using file handles or file pointers as in older languages based on C, file operations use an object.

SS file operations use a hierarchy of object types in opening a file. These ensure that reading a file can only be done on a file open for reading, etc. There are objects for each type of I/O.

The definition of the file operation functions and modes and object types is in standard library member **io.sslib**.

The hierarchy of object types and what they can do is shown in outline form:

Table: File Operations Object Type Hierarchy

Object Type:	Creating Modes:	Definition:
_File	*string*.file	object identifying the name/path *string*
_TextFile	_*File*.text	object operating on text lines in a file
_ReadTextFile	_*TextFile*.input	object reading text lines, file exists
_WriteTextFile	_*TextFile*.output	object writing text lines, replace a file
_WriteTextFile	_*TextFile*.append	object writing text lines, end of existing file
_BinaryFile	_*File*.binary(object *blk*)	object operating on a binary file of fixed sized blocks, the size of an object *blk*
_ReadBinFile	_*BinaryFile*.input	object reading binary blocks, existing file
_WriteBinFile	_*BinaryFile*.output	object writing binary blocks, replace/new file
_UpdateBinFile	_*BinaryFile*.update	object reading or writing binary blocks

This object type file system limits operations to appropriate functions and values, capable of being checked at compilation/syntax checking time, reducing errors.

Each subsidiary type inherits the prior types, and some of their functions remain available.

From the above hierarchy, we see that a given object type may be only an intermediate step, and we see that the modes input, output, append and update set a working_mode and return a type, as appropriate. Once open, the working_mode cannot be changed.

A binary file is a series of fixed-size blocks which may be buffered or combined. It can also be opened in update mode, which allows randomly positioned reads and writes.

The above use standard error handling - error condition codes are TBD.

Open Files

An *open_file* is an object of one of these types:

_ReadTextFile, **_WriteTextFile**, **_ReadBinFile**, **_WriteBinFile**, or **_UpdateBinFile**.

An *open_file* has a type which enables the interface **_OpenFile**.

Table: Interface _OpenFile Procedures

Name:	Action:
close	Closes the file

Table: Operations on an Open File

Object Type:	Procedure/Property:	Description:
_ReadTextFile	read	property, returns a string (a line) or **null**
	rewind	mode, sets position to first line
_WriteTextFile	write(string *str*)	function, writes str, returns logical
	writeLine(string *line*)	function, writes line with \n added, returns logical
	writeLine(string lines[])	function, writes all lines with \n added, returns logical
_ReadBinFile	rewind	mode, sets position to first byte (number 0)
	read(ulong *blknum*)	returns a string, one block, at data block number *blknum*, or **null**
_UpdateBinFile	read(ulong *blknum*)	returns a block, at block number *blknum*, or **null**
	write(ulong *blknum*, object *blk*)	function, writes a block, to block number *blknum*, returns logical
_WriteBinFile	write(struct *blk*)	function, writes a block, to next block number, returns logical

Table: Additional File Property

Function:	Base Type:	Result:	Description:
get_lines	_TextFile	string array	property; opens, then reads a text file and returns the lines as an array of strings with end-of-line characters trimmed off, then closes the file

Example of Text File Reading

```
# Open and read a file named 'data' and stops if the word 'cloud' appears - -
_ReadTextFile inp = 'data'.file.text.input
```

```
for while (string line = inp.read) # stops on null return
{
   if line.find('cloud') >= 0
   {
      printLn "Word 'cloud' found in line {line}."
      exit
   }
}
print "Word 'cloud' not found."
```

Another Solution

```
# this is an alternate solution to the same problem - -
# this is much shorter, but it can run for a longer time
if 'data'.file.get_lines.Join('*').find('cloud') >= 0
{
  print "Word 'cloud' found"
}
else print "Word 'cloud' not found"
```

Date and Time Processing - Unfinished & Postponed Features

There are many issues in date processing because there are multiple formats to be supported, and because dates are related to localization issues.

The primary internal format is the Unix timestamp, which is actually the number of seconds since the start of 1970, as a 32-bit integer. Windows counts since 1980 instead.

In addition, there is a microsecond counter available in Unix.

Another format commonly used in Unix systems is the "tm" format, traditionally an array of integers, representing hours, minutes, seconds, day, month, and year. This format is easy to work with, but the order of these integers is prone to error in writing programs. Another complication is that months start with 0, days with 1.

Dates and times and timestamps are yet another set of formats when they are used with MySQL databases. Timestamp formats in MySQL changed after a certain release of MySQL.

Then there is the issue of external formats. American usage differs from other countries, which also differ with each other. This issue is also complicated by languages - how to spell the months and days.

There are external standard formats, RFC 822 for Internet, RFC 1123 time format, ISO 8601 standard format, which is not specific, and often not followed.

Additional issues include time zones, Universal Time, GMT, daylight saving time adjustments, and 12 hour vs. 24 hour formats.

Consequently, this area will be unspecified in the early stages of specifying the SS language.

MySQL® Database Operations

The standard library member **mysql.sslib** defines objects and functions which enable queries and operations on MySQL databases.

The central object which is used has type _MySQL, and it is used to open a database:

```
_MySQL db
# modes user and pwd set necessary values to connect to the database - -
# they can be used in any order
db.user('userid').pwd('password').connect # - - to the default database

# When the above variable db is used with the query function,
# it returns an object - -
_MySQLresult result = db.query("SELECT * FROM mytable")
```

Table: _MySQL Procedures

Procedure:	On Type:	Description:
user(string uid)	_MySQL	Mode, setting user id before connecting
pwd(string pwd)	_MySQL	Mode, setting password before connect
defaultdb(string db)	_MySQL	Mode, changing default database before connect
connect	_MySQL	Property, opens the database, returns logical
close	_MySQL	Method, closes the database and clears connection data
isConnected	_MySQL	Property, returns logical
query(string sql)	_MySQL	Function, returns object type _MySQLresult
database_names	_MySQL	Property, returns array of strings
table_names	_MySQL	Property, returns string array of tables in default db
table_names(string db)	_MySQL	Function, returns string array of tables in database db
table(string tbl)	_MySQL	Function, returns object type _MySQLtable
fetch	_MySQLresult	Property, returns an array with string index for a row
fetchAll	_MySQLresult	Property, returns an array of arrays (rows)
Count	_MySQLresult	Property, returns type int, number of rows returned or affected by the query

The fetch function is valid after SELECT or SHOW, but not after INSERT or UPDATE, etc.

After a fetch, the next row is obtained. Each row is a string array with string keys. Each field is a key (an unnamed or expression column gets a pseudo-name from the query) with its string value. Each call to fetch gets a row or returns **null**, to indicate no more rows.

Object type _MySQLresult functions use standard error reporting - error condition codes are defined.

Object _MySQLtable

This object describes a table, in simple form, and allows access to its fields.

Table: Object _MySQLtable

Property:	Description:
table	string, name of the table
num_fields	number of fields
fields	array of objects of type _MySQLfield, keys are the string field names
primaries	string array, names of the fields in the primary index

Object _MySQLfield

This object defines the attributes of a field.

Table: Object _MySQLfield

Property:	Description:
name	string, name of the field
type	string
valuea	string array
extras	string
quoted	logical - indicates whether values must be quoted

The valuea item depends on the type. If it is 'enum' or 'set', it is a list of values, for example. For simple types, it is an array with one element, the default value.

Extras may indicate an integer is UNSIGNED. It will indicate the default and the null behavior. This string depends on the type.

These and 'type' depend on the values defined by MySQL.

Formatting Strings

A set of built-in functions, modes, and properties are defined in standard library member **fmt.sslib**. These are used to format strings for printing/output.

These modes are (mostly) based on object type **_Fmt**. A new text string is started with the function 'Fmt', no base object, creating an empty string, adding text elements by appending. Each of these functions and modes begin with (or are a single character) an upper case letter. Then at the end, a last property is 'P', which yields the string result.

Some of the modes do not add text; they set parameters for a following value. These are the *parameter modes* W, D, Z, C, R, and L. A value mode must follow one or more of these modes. Mode D also works for integers, in effect dividing by 100.0 for 2 decimal points, etc.

Modes R and L establish justification. Right (R) is default for numeric values, left for strings.

After any value item, the modes W, A, P, D, E, Z, C, R, and L are reset. Modes P and Z are mutually exclusive.

Table: Built-in Formatting Functions, Properties and Modes

Procedure:	On Type:	Description:
Fmt	(none)	Mode, establishes a new _Fmt object
P	_Fmt	Property, returns the produced string (“Print” or “Produce”)
X(int n)	_Fmt	mode, adds n blanks
W(int width)	_Fmt	mode, establishes width of next item
O	_Fmt	mode, resets all options to default
C	_Fmt	mode, shows thousands commas on N following
M	_Fmt	mode, shows negative numbers in parentheses, and adds a blank on the right for plus values. Cancels Z
L	_Fmt	mode, sets left justification (default)
R	_Fmt	mode, sets right justification
Z	_Fmt	mode, sets leading zeros for N (int) item following, cancels M
EOL	_Fmt	mode, adds end-of-line characters
D(int d)	_Fmt	mode, sets **d** decimal places for numeric item following
E	_Fmt	mode, toggles mode for N (float) following to either display exponent style rather than fixed point or resume fixed
S(string str)	_Fmt	mode, inserts the string value str, obeys W, R, L modes
N(n)	_Fmt	mode, inserts the numeric value n, obeys W, R, L, D, C, M, and Z modes. Ignores E mode if not float. Disallows M with Z mode. Float n ignores Z mode and C mode, and E mode implies a request for scientific mode

The D mode sets a number of decimals shown on a float or integer value.

Scientific mode has an adjusted exponent, to a multiple of 3, and it adjusts the number of digits before the period to 1 or 2 nonzero values as needed. Applies to float only.

If a value cannot be converted, a string of '#' characters is output.

This formatting system is designed to optimize output and also enable compiler error checking.

It is less error prone than systems based on C language fprint.

The value of a stack of modes on Fmt ending with P is the formatted output as a string.

Examples of formatting

```
int tests[], ic = tests.Count   # assume the tests array has been filled somehow
print Fmt.S('For ').C.N(ic).S(' tests, the average is ').W(12).R.D(3).N(tests.sum/ic).P

int Nused = -3547822
int plusV = 78845
string disp1 = Fmt.W(12).C.M.N(Nused).P     # sets value of disp1 to: " (3,547,822)"
string disp2 = Fmt.W(12).C.M.N(plusV).P      # disp2 is set to:              "     78,845 "
string disp3 = Fmt.W(8).Z.N(plusV).P          # disp3 gets value:         "00078845"
```

Localization of Formatting

The pragma statement can control some options in formatting:

pragma locale *option*=*value*, ...

where *option* (not reserved names) and *value* can be:

Table: Locale Options

Option:	Value:	Description:
thousands	',' (default) or '.' or ' ' or '' or '_'	separate every 3 digits in integer part
decimal	'.' (default) or ','	decimal point in input/output

Regular Expressions

Regular expressions are implemented in SS using PCRE, "Perl Compatible Regular Expressions."

Regular expression procedures operate on a string, either changing the string, or finding values in the string, or matching and validating the string.

In order to do any regular expression procedure on a string, the first step is to apply the function RegExp on a pattern string, which yields an object type named **_RegExp**, which has several modes and functions which can be applied. For example, the function match returns true or false indicating the success or failure of a match.

Example:

```
if /def$/.RegExp .scan("abcdef").match { printLn "Found 'def' at the end." }
```

Table: Built-In Procedures/Properties for Regular Expressions

Procedure	On:	Description
RegExp	string pattern	Property, apply pattern string, creating a _RegExp object
limit(int *lim*)	_RegExp	Mode, sets a limit of replacements. Default (-1) is no limit
prefs(_PCRE *pr*)	_RegExp	Mode, sets preferences to *pr*, a nameset _PCRE
match(string *str*)	_RegExp	Property, returns logical if the pattern matches string *str*
replace(string r[])	_RegExp	Function, returns the string *str* from scan, with matches replaced from the array *r*
extract(string *str*)	_RegExp	Property, returns an array of values found by the pattern in string *str*
Remove	_RegExp	Property, matched parenthesized values are deleted in the returned string
numChanged	_RegExp	Property, int return, gets number of replacements made

Regular expression patterns use curly braces. In a constant, begin and end with a slash or apostrophes for pattern constants. Watch for embedded {, }, or \ if using quotation marks.

First, apply the desired modes, then an action function. The order these modes are applied does not matter. Count is used after an action, on a saved _RegExp object. It is not stacked on the action.

The **_PCRE** flags ?? type defines "option" values from a **nameset PCRE_Names** with:

PCRE_CASELESS	PCRE_UNGREEDY	PCRE_PARTIAL
PCRE_ANCHORED	PCRE_NOT_BOL	PCRE_NOT_EOL
PCRE_DOTALL	PCRE_DUPNAMES	

Optimization of Regular Expressions

The pattern property "compiles" the pattern, saving an optimized form in the _RegExp object. This object can be created outside a loop, saved, and used to do matching or other procedures inside the loop. This separates the regular expression task into optimized parts. For example:

```
_RegExp zipPat = /^\d{5}$/.pattern  # match a 5-digit zip code
for # ... etc. - going through multiple zip code inputs
{
   if zipPat.scan(aZipCode).match
   {
      printLn "success!"
   }
}
```

This style of usage speeds up repetitive applications of a single pattern, especially a pattern with some complexity. A pattern string (with slashes) is precompiled by the SS compiler.

HTML and URL Functions

A number of functions that handle special characters passed through the Internet are defined in standard library member **std.sslib**:

Table: HTML and URL Functions/Properties

Function/Property	On:	Description
stripHTML	string	Return string with all HTML or XML tags removed
stripHTML(string array h)	string	Return string with all HTML or XML tags removed, except keeping the tags in the array **h**, case independent
HTMLspecials	string	Return string changing apostrophe, quote, &, > and < to entities
HTMLspecialsDecode	string	Return string, with reverse of HTMLspecials conversion
URLencode	string	Return string, replacing blanks with + and ampersands and apostrophes and quotes with %xx?

Complex Data Type

This sample portion of the complex type definition is taken from the standard library member **complex.sslib** which implements the object data type complex, with associated operators and functions. Part of the source for that library is shown here.

Part of 'complex.sslib'

```
# define complex arithmetic and functions - - -
set object _Complex
{
   private float Re, Im # complex type is immutable
   float Real return Re, Imaginary return Im # read-only properties
# constructor - - -
   default mode (float Real = 0, float Imaginary = 0) # with defaults
   {
      Re = Real
      Im = Imaginary
   }
# - - - define arithmetic operations
# etc. - - -
# - - - addition
   mode +(_Complex y) return _Complex ans
   {
     ans.Re = Re+y.Re
     ans.Im = Im+y.Im
   }
# etc. - - -
   # complex arctan
   public function arctan return _Complex
   {
      _Complex a = _Complex(Re, Im + 1.0) # add "i"
      a /= _Complex(-Re, 1.0-Im) # a = a / ("i" - this)
      return (0.5 * a.log).TimesI
   }
# etc. - - -
   # complex arctanh
   public function arctanh return _Complex
   {
      return -(this.TimesI.arctan.TimesI)
   }
} # end of object definition
```

Example - Polymorphism

This example is patterned after a set of sample programs in an article on polymorphism found online. It shows how interfaces and inheritance interact.

The example shows an object type named Animal, inherited by objects named Cat and Dog:

```
interface SaysWhat # implement an interface - - -
{
   string Name set return
   method Speak
}
# the base object Animal - - -
Animal object enables SaysWhat
{
  string animal_name
   string Name set animal_name = Name return animal_name.title
   method Speak
   {
      printLn "{Name} says \"I am an animal.\""
  }
}
# animals Cat and Dog
Cat object inherits Animal # overloading Speak for Cat - - -
{
   method Speak
   {
      printLn "{isType} {Name} says \"Meow!\""
  }
}
Dog object inherits Animal # overloading property Name for Dog
{
   Name set animal_name = Name return “Canine: ” + animal_name.upper
}
# the program - - -
Cat Fluffy = Cat(Name='FLUFFY'), Tom = Cat(Name='tom cat')
Dog Fido = Dog(Name='Fido')
Animal pets[] = Animal[Fluffy, Tom, Fido] # an array
for each who = pets who.Speak
# expected output -
# Cat Fluffy says "Meow!",
# Cat Tom Cat says "Meow!"
# Canine: FIDO says "I am an animal."
```

Extended Example - Binary Search Trees

Create the interface and object definitions needed for creating binary search trees, or _BSTs. These are defined as binary trees (two leafs per node, a left and right subtree) where a left-to-right (in-order) walk yields the values in ascending order. The trees in our example do not permit duplicate key values, even though many standard definitions of _BSTs allow duplicates. In addition, we will provide several utility procedures.

This example allows _BST trees of any object type as data.

Interface _BST

The interface is parameterized, as follows:

```
interface _BST(key_type: _Comparable \ float, data_type: *) enables _Traversable
{
  self Left, Right # protected
  key_type Key # protected
  public data_type Data
  # _Traversable - - -
  self First return
  self Next return # supply a 'next item' on the object
  [key_type k] # indexer - - - so it is an array
  mode Insert(key_type k, data_type d)
  mode Clear(key_type k) # deletes it
}
```

Note that a float value cannot be an index. All integer types and string types are allowed, as well as object types that implement the _Comparable interface.

Object Type '_TreeNode'

Define a tree node named _TreeNode, using the interface, creating trees with uint, int or string key, and a method which inserts a node. Implicitly enables _Traversable interface.

By inheriting this type and adding data to the nodes, you implement a binary search tree for any simple key. This can be used for a dictionary or symbol table. When the tree is balanced, searches are optimal.

```
object _TreeNode (key_type: _Comparable \ float, data_type: *) enables _BST
{
    self Left, Right
    key_type Key # kept in order on this member - protected access
    public data_type Data

    # a function which finds the Last (rightmost) key:
    public function Last return key_type ans
    {
      for node = this while node.Right.isNotNull repeat node .= Right final ans = node.Key
    } # implicitly returns ans

    # needed to implement _Traversable interface - - -
    # a function which finds the First (leftmost) key:
    public function First return key_type ans
    {
      for node = this while node.Left.isNotNull repeat node .= Left final ans = node.Key
    } # implicitly returns ans
```

function which gives the number of nodes in the tree:

```
    public function Count return int
    {
      if isNull return 0
      return Left. Count + Right.Count + 1
    }

#   self Next return TBD
```

```
  # mode 'Insert' - add a new node with a key and data, returning it.
  # if the key exists, change the data
  public mode Insert (key_type k, data_type d)
  {
    _TreeNode newNode = _TreeNode(Key=k, Data=d)
    if Key =? k
    {
      Data = d # alter action if unacceptable
      return this # already there
    }
    else if Key > k
    {
      if Left.isNull
      {
        Left = newNode
        return Left
      }
      else return Left.Insert(k)
    }
    else
    {
      if Right.isNull
      {
        Right = newNode
        return Right
      }
      else return Right.Insert(k)
    }
  }
```

A mode which finds a node with matching key, returning null if not found:

```
  public mode Find(key_type k)
  {
    if isNull return null
    if Key =? k return this
    if Key > k return Left.Find(k)
    else return Right.Find(k)
  }

# the indexer - -

  [key_type k] return Find(k) # an indexer acts like a mode
```

```
# a method which deletes a node by the key value - - - (UNFINISHED)
# interface requires it
# start at the root
# does nothing if key is not found

        public mode Clear(key_type k) return pivot
        {
          self pivot = this # a way to look back
          if k =? Key # delete the present node
          {
          }
          else if k < Key # go down left
          {
          }
          else # go right
          {
          }
        }
```

```
# function Equals returns true if two trees are structurally identical,
# independent of the key_type and data_type used,
# comparing keys and data
# internal references are not compared:

    public function Equals(self T2) return logical
    {
      #1: both empty
      if isNull And T2.isNull return true

      # 2.  both non-empty, compare them:
      else if isNotNull And T2.isNotNull # use recursion
      {
        return Key =? T2.Key AND Data =? T2.Data  AND \
               Left.SameTree(T2.Left) AND Right.SameTree(T2.Right)
      }

      # 3.  one empty, one not:
      return false
    }

# A  function Depth returns the maximum root-to-leaf depth of the tree:

    public function Depth return int
    {
      if isNull return 0
      else return Left.Depth.Max(Right.Depth) + 1
    }

# This is a generator function.
# It returns each tree key in key order, independent of key type, doing an In-Order Walk

    public function Walk repeat return key_type
     {
       if isNotNull
       {
         Left.Walk
         return Key
         Right.Walk
       }
       return null
     } # end generator
```

```
# A  function which returns the tree keys in left-right key order (“in order”) as an array,
# independent of the key_type and data_type used:

    public function Keys return key_type vals[]
    {
        for each node = this.Walk { vals[] = node.Key }  # use the generator Walk
    } # returns vals array

# function which right-rotates the tree at a node. This method needs to know the tree's root.
# It is independent of key_type. This method can be used in balancing a _BST.
# The  base is the whole tree root or a subtree root.
# These rotations are 'protected'

    mode rightRotate return self pivot # 'this' is the old tree root
    {
      if Left.isNull return # cannot do anything
      pivot = Left # declare the new subroot
      Left = pivot.Right
      pivot.Right = this
      # now pivot is the new root, old root is right child
    }

# The complement function, doing a left-rotate. Same conditions. Note the similarity.

    mode leftRotate return self pivot # 'this' is the old tree root
    {
      if Right.isNull return # cannot do anything
      pivot = Right # declare the new subroot
      Right = pivot.Left
      pivot.Left = this
      # now pivot is the new root - -
    }
```

```
# This mode is applied to the tree once it is built, or after insertions. It ensures any subtree is
# exactly balanced or the left branch depth is at most 1 more than the right branch depth.
# This optimizes finding a value since the left side is checked first.
# This is called a left-leaning balanced tree.
# The tree is not kept in a balanced condition until this method is called.

  public mode Balance return self pivot # base is subtree or root
  {
    if isNull return this # nothing to do
    pivot = this
    int diff = Left.Depth - Right.Depth
    # choose a case - -
    switch diff
    case 0,1 # good enough
      break # no balance needed this node
    case 2: # left side is too deep (diff is 2 or greater)
      for while diff > 1 repeat diff-- { pivot = pivot.RightRotate }
    default # negative - right side is too deep
      for while diff < 0 repeat diff++ { pivot = pivot.LeftRotate }
    end # switch diff
    pivot.Left.Balance
    pivot.Right.Balance
    # returns pivot
  }

} # end of the object def
```

_TreeNode Example

Use the definition of _TreeNode - - build a simple tree with values 1, 2, - - random order

```
_TreeNode(key_type:int,data_type:string) root = _TreeNode(Key=3, Data='three')
root.Insert(Key=2, Data='two')
root.Insert(Key=1, Data='one')
root.Insert(Key=4, Data='four')
root.Insert(Key=5, Data='five')
root .= Balance

# use function Keys - - -
int keys_used[] = root.Keys # using our tree
# print the values - - -
printLn "values: {keys_used.Join(',')}" # prints "values: 1,2,3,4,5\n"
```

Object Type _BalancedTree

The following definition uses _TreeNode to create a left-leaning balanced tree. This is the object model for non-fixed arrays in SS.

```
Object _BalancedTree (key_type: _Comparable \float, data_type: *) inherits _TreeNode
{
   mode Insert (key_type k, data_type d)
   {
      parent.Insert(k,d)
      return this.Balance
   }
}
```