The SS
(Scripting System)
Programming Language

Version: February 22, 2019

Table of Contents

Author and Copyright

Copyright © 2009-2019 - All Rights Reserved - John T. Bagwell Jr. of Sandpoint, Idaho

The author reserves all rights to the SS language concepts introduced in this document. Please request permission before quoting any part.

The SS Programming Language

SS is a language designed for general application programming or server scripting. It is a complete programming language, not just for scripting, designed to be compiled or 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 capable of being 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. Output strings to text files and database values and printed strings are by default in UTF-8 encoding. Support also is supplied for UTF-16 and UTF-32 and the single byte codes ASCII and the ISO-8859-1 encoding.

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

Some of the Features in SS

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

- All variables and arrays are declared and typed. It is a statically typed language, meaning variables and functions (etc.) are explicitly typed. It is a weak typing language, meaning there are implicit type conversion rules, not requiring type casting for built-in types.

- 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.

- The case of reserved words is lower case, except the six reserved words and, or, not, true, false, null can also be upper case: AND, OR, NOT, TRUE, FALSE, NULL.

- 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.

- Another concern is maintainability. Punctuation using braces is avoided since it leads to need for corrections during development, and the braces are not consistently used by people who prefer different styles.

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

- 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 "square" arrays are allowed.

- A number of array procedures are provided.

- The language is expandable. For example, type complex is a defined object type, an existing definition is provided.

- The **code** statement allows special codes to be used in strings.

- Names can use additional European characters and accented letters as letters.

- Pointers are not used. In some cases a choice between value and reference is provided.

- 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 "**==**" and inequality is "**<>**" rather than "**!=**". Semicolon is not used.

- The word **new** is not required.

- 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 may just set an indicator or ithe settings of an object. It may return a new object reference.

- A generator function enables obtaining a sequence of values.

- A list or tree can be made to operate like an array.

- Functions can support multiple types with a single definition.

- A function or mode or method invocation with no arguments does not require a pair of parentheses.

- An enhanced object type can keep units apart, like Fahrenheit versus Celsius, or Meters versus Inches.

- The model for file manipulation is different, more object-oriented.

- 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 and subscripting and type casting can be defined.

- Variant procedures provide an alternative to abstract procedures in an object type definition.

- Some reflection features are included.

- A simpler condition-handling **raise** and **when** are used instead of error handling in the **try**, **throw**, **catch** style.

Omitted Language Features

These are some of the omitted 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.

- Garbage collection is used automatically.

- Structs.

- Type punning, or union. Omitted because of danger and misuse.

- Namespaces, modules, packages, etc. Simple scope rules are used.

- Multiple statements per line.

- Events and event handling.

- Threading and parallel computation.

- Prefix ++ and -- are omitted; postfix ++ and -- are included.

- 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. No graphic support. Programs run on a server or as background tasks..

- Closures, functions as arguments, callbacks, lambda expressions. Procedure variants supply some of these capabilities.

Some Style Rules and Suggestions

- A script consists of object definitions, interface definitions, shared definitions and pragma statements.

- There are no executable statements, control constructs, or assignments outside of an object.

- The main program is defined in the object named @Main, the constructor is the program.

- There are no multiple statements on a line.

- Comments follow a # sign, to the end of the line. The comment mark 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, operators using multiple characters and numeric constants cannot be broken. The reverse slash is not effective as a continuation mark 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 have value \n.

- Programs are expected to use indentation for readability. Readability suggestions –

  ```
  Surround assignment operators with blanks.
  Place a blank after a comma.
  Indent contents by at least 2 or 3 positions.
  Keep source lines shorter by using continuation.
  Use lots of comments.
  ```

- 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 or regular expression patterns:

  ```
  ` grave accent
  ; semicolon
  ! exclamation mark
  ```

- Braces are used only in strings for embedding values and hex values and patterns:

  ```
  { left brace
  } right brace
  ```

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 directory **user/** with extension **.ss**. There are some standard library files which are automatically included, in directory **std/**. These define standard objects or procedures, interfaces and constants. The directory names are not reserved.

The *library_spec* has the form: **directory/member** where **directory/** can be omitted; a search is then done for the member in the directory **user/** then the directory **std/**. Subdirectories are permitted.

The *library_spec* cannot be a string expression. Quote marks are not used.

This pragma is not allowed inside objects, procedures, or any definition.

Apache server can be set to run any default file as the main file. The name **main.ss** or i**ndex.ss** is suggested.

Standard libraries are implicitly included, as needed. They are:

Table: Standard Libraries

Name (.ss):	Defines:
std/array	Array functions and properties
std/codes	Code names
std/cond	Conditions
std/config	Configuration file
std/directory	Directory functions, properties
std/files	File functions, properties
std/fmt	Formatting output
std/form	HTML and form input definitions
std/io	File operations
std/math	Mathematical constants and functions
std/mysql	MySQL functions, objects, properties
std/regexp	Regular expressions
std/std	Miscellaneous procedures etc.
std/string	String functions and properties

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 <u>class</u> 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 emtity 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 of basic types. 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.

- *basic type* – the predefined types **int**, **uint**, **float**, **string** and **bool**.

- *shared definition* - the language assumes everything is an object, with some built-in properties and procedures. A **shared definition** is a way to add baseless methods, functions on basic types, nameset types and constant names.

- *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, do, each, 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 in a construct. A block acts as a statement.

- *procedure* - a method (a "subroutine") 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.

- *generator function* - a function which behave like an array in an *each* statement.

- *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 of the object and 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 created pointer.

- *value type* - a type which is passed by copy. The types uint, int, bool, float and nameset.

- *code* – a character code for strings, specifying character size and values.

Reserved Words

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

Reserved words are in lower case, except **and**, **or**, **not**, **null**, **false**, **true** may also be in upper case for readability, as **AND**, **OR**, **NOT**, **NULL**, **FALSE**, **TRUE**. Other case forms of these keywords are not reserved words, but should be avoided.

Reserved words are also called keywords. There are 66 reserved words.

Table: Reserved Words

abstract	continue	FALSE	mode	parent	set	ulong
and	default	final	nameset	partypes	short	until
AND	do	float	not	pragma	static	ushort
bit	each	function	NOT	private	string	variant
bool	else	if	null	public	switch	when
break	elsif	inherits	NULL	raise	this	while
byte	enables	int	object	ref	true	
case	end	interface	of	repeat	TRUE	
code	exit	long	or	return	ubyte	
const	false	method	OR	self	uint	

Names

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

Initial underscore is used for system-defined names.

Letters are the 26 English letters A through Z, upper or lower case and selected Western European letters and accented letters from the Unicode set, which have a single byte value from 0xC0 (192) to 0xFF (255), as follows:

```
European 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
European 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
Additional European lower: ß ÿ
```

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

Invalid names: 1A, two words, A&B, x-hyphen, ¥999.

Configuration

The configuration data is found in library file **std/config.ss**. It consists of a pragma line:

```
pragma config
```

followed by *command* lines of this form:

```
command value1 value2 …
```

For example, it contains these lines:

```
locale thousands ','
locale decimal_point '.'
```

Main Program

A "main program" begins by implicitly constructing and instantiating an object with the *type_name* @Main, in effect, executing a line:

```
@Main
```

This invokes the constructor of an object type which is named @Main.

Blocks and the Scope of Names

A *block* is a sequence of statement and declarations which delimit the scope of meaning for names. A block is also called a *construct*, since it usually consists of multiple lines.

A block begins with a keyword and ends with an **end** statement.

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 and not visible.

Within a block, a name cannot be redefined except in a new scope level, a nested block.

All names defined outside an object have global scope, available anywhere.

Names of object types, nameset types, partypes names and interface names are in the same name 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, unsigned integer, float, bool, nameset type names and string. Except for nameset type names they are also called basic types.

New types can be added as objects. Type **complex** has an available object definition.

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.

Table: Integer Value Ranges

Type	Alternate Name	Integer value ranges, shown with commas
uint.1	bit	0 or 1
uint.8	ubyte	0 to 255
uint.16	ushort	0 to 65,535
uint.32	Uint	0 to 4,294,967,295
uint.64	ulong	0 to 18,446,744,073,709,551,615
int.1		0 or 1 (unsigned)
int.8	byte	-128 to 127
int.16	short	-32,768 to 32,767
int.32	int	-2,147,483,648 to 2,147,483,647
int.64	long	-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 shorter (standard float) or longer (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 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**. The maximum is approximately **1.7976931348623157e+308**.

Bool

The type **bool** is a logical value, with true or false values. It is named for George Boole, who defined logic manipulation rules. It is also called Boolean or LOGICAL in some programming languages.

The result of a comparison or bool expression is either **true** or **false**. A bool value occupies 1 byte in storage. An array of bool values packs the values into bits if the index is a nameset type or fixed size and the maximum number of required bytes is 256. True has value 1 and false has value 0 internally.

The **if** statement and other places require a bool expression. The basic arithmetic types may be type cast to **bool**. A nonzero value becomes **true**, a zero or **null** value becomes **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, by default, using 8-bit bytes. Strings are immutable; internal values cannot be modified.

The **code** statement allows a different encoding to be selected. In a string declaration, a code name can be specified after the keyword **string** with a period, or for an individual item after the name. Example:

```
string.UTF16 abc, def.UTF32
```

There is no separate type for single characters.

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

Nameset

Nameset items are named constant unsigned integer values, grouped into a new type name. This is similar to the C language 'enum', but more restrictive because the names are typed. A nameset constant can be used only in a variable or array using the correct nameset type name as its type, or as an index for an array with the same nameset type index.

Complex

Complex type is available as a defined object type named **complex**. It is a pair of numbers, one 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 definition is found as an object type in the library **std/complex.ss**. It is NOT automatically included. The type name **complex** is not a reserved word. Alternative implementations can be defined.

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 or basic types, and namest typed values.

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.

Expression Operators

Table: Operators in Expressions

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

Highest priority numbers are evaluated first.

The right operand of a shift operator is an unsigned integer, less than 64 in value.

The assignment operator **=**, the *op=* assignment operator and **.=** have a value, the same as the left operand after assignment. They associate right-to-left, which permits assignment to multiple variables:

abc += ijk = xyz .= abs

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, mode or return-type property name. The left side is an Lvalue, the right side function or property is applied to the left side.

Examples:

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

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 **bool** 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.

The selection period must not have spaces around it, except the operator **.=** may have a blank before the period.

Postfix Operators ++ and --

Postfix operators ++ and -- require the left operand to be an Lvalue. An object reference to an element completes the reference, returns that value, then increments it.

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

There are no prefix versions of these operators.

These can be statements because they alter a value.

Unary Operators

The unary operators **+**, **-**, and **~** are restricted in placement. They may not follow a binary or unary arithmetic operator. They may follow a left parenthesis, a left square bracket, an assignment operator, a comma, or the logical operators **AND**, **OR**, **NOT**, **and**, **or**, **not**.

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 **|** do not short circuit.

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 **bool**, **int**, **uint**, **string**, **float**, or a nameset type name or an object type name.

An initial value can be assigned:

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 bool 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 by type casting, 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 and a value too large to be an int both raise a conversion condition.

A string value like "123GO" does not convert to the integer 123. It raises a conversion condition.

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.

Bool Expressions

A **bool** 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-bool value cannot be compared with **true** or **false**. Bool values cannot be compared with =? or <>. The same result is obtained by use of **AND**/**and**.

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

A bool expression can be cast into an integer value; **false** will be 0 and **true** will be 1.

The bool operators **AND** (**and**) and **OR** (**or**) in a bool expression evaluate using shortcut evaluation. If a value can be determined while skipping a subsequent portion of the expression, the skip is permitted. This shortcut evaluation does not apply to the bitwise and (**&**) and bitwise or (**|**) operators.

One example:

```
if abc <= 9 AND def > 0 # the part "def > 0" is skipped if abc is > 9
    break
end if
```

Null

The value **null** (or **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**.

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 <u>not</u> "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. A type cast is implicitly also a call to the constructor.

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 to:	uint	int	float	bool
uint	Yes	#1	Yes	#2
int	#1	Yes	Yes	#2
float	#3	#3	Yes	#2
bool	#4	#4	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. From **int** to **uint** may convert a sign to a data value, and vice versa.

#2: The result bool value is **true** if the base is nonzero, **false** if it is zero.

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

#4: The bool value becomes 0 if **false**, 1 if **true**.

A value **null** cannot be type cast except to a **bool** value of **false**.

Data Values

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

- A simple item with a type bool, float, int, uint, a nameset type or string.

- An array, which has simple integer indexes, nameset indexes or string values as indexes. The first index is 0 when unsigned integer 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, which has a type.

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 numeric 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.

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

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:

[**=** *array_constant*]

The *type_name* is omitted and implied on this initial constant.

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**.

An array index cannot be type **float**, **bool** or an object type.

The use of *nnn* implies a *fixed size* array of *nnn* members, where *nnn* is a nonzero positive integer constant or integer constant 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.

More examples:

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

An array index has integer values, nameset values, or string values. 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:

```
const uint msiz=100
float scores[msiz] # a fixed size array
int cards[52]
float table[10][string] # a fixed size array of string-indexed arrays
table[5]['chair'] = 236.98
```

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 4 array elements, indexes 0,1,10,11
```

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

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

```
float salary[string] = ['driver'=12_000.00, 'sales'=15_000.00]
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 a '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 non-fixed-size 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

Array elements are indicated as follows:

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

An array with nameset values as indexes can use its array index in square brackets as usual:

```
Options[CameraAvailable] = TRUE
Prices[Truck] = 35_000
```

or if the index is a nameset constant name it can use an object member notation with the index name after an object selector period:

```
Options.CameraAvailable = TRUE
Prices.Truck = 35_000
```

A missing array element reference 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 # where abc is fixed-length and def is also, but different length, assigns def up to the length of abc, but if def is shorter, default values are assigned to fill abc. Zeros for arithmetic values and namesets, zero-length for strings, nulls for other objects.

- 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*= def # is equivalent to: abc = abc *op* def, where def is an array or a scalar value.

- 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 **.**= *proc* # where *proc* is a property or procedure (function, method, mode), applies *proc* to each member of abc.

- 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 Values and Array Constants

An array value is a sequence of appropriately typed values enclosed in square brackets, separated by commas. A final comma may appear with no value following. Index values may be supplied in the list followed by equal sign and value. All the values must be the same type, and all the indexes must be the same type, with types matching the array definition.

An array of arrays will have array values in each position of the list, as required.

An empty array is designated as a pair of square brackets, **[ ]**. Spaces inside are optional. The type can be specified left of the left square bracket, or it is assumed to be the required type.

A nameset array value is written like any other array value, as a list of the nameset value names as the value list. All member names must be from the nameset.

If all the values are constants, the bracket-enclosed array value is an array constant.

Example:

```
float prices[string] = ['pencils'=0.99, 'eraser'=0.69,]
uint bedrooms[] = [3, 1, 4, 2]
```

The type of the array element values and indexes in an array value or constant is assumed, in most cases, from context. An initial value assumes the type being declared, and a value passed as an argument or assigned to an array assumes the expected type.

If desired, or if the type is not obvious, the type of the array can be placed just before the left bracket. The index types must all be the same and the type cannot be specified.

Table: Built-in Array Functions, Properties and Methods

Name:	Returns:
clear	method, empty array, delete array element
count	the number of elements
firstIndex	lowest index in the array
flip	exchanges indexes and values of an array. Duplicates are lost
getIndexType	string property – type name or 'fixed index', implying uint
indexes	array of index values used for elements of an array
join(string sep)	string, concatenated with separators sep, on a string valued array
lastIndex	the highest index of an array
maxValue	maximum value in an array
minValue	minimum value in an array
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. Argument val can be a variable or expression
reverse	a new array, integer indexes, values in reverse order
slice(uint m,uint n)	an array extracted from index **m** through **n**, **m** <= **n**
sort	a new array, integer indexes, sorted
values	array of values ignoring the indexes. Integer indexed result

Built-in array procedures are defined in standard library member **std/array.ss**.

String as an Array

A string value is partially usable as an array of one-character bytes, or a character of an appropriate size for the code used. The bytes are indexed from 0 like an array. Indexing a string in an Lvalue is not allowed since strings are immutable.

Note that a Unicode code point in UTF-8 or UTF-16 may occupy several bytes.

When a subscript is used, the result is one byte for UTF-8, and 2 bytes (a **uint.16**) for UTF-16, etc. The character size can be controlled with the **code** statement.

Nameset Types

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

```
nameset nameset_name name [ = unsigned_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 type, 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.

The value names cannot be assigned into a variable using a different nameset type.

Nameset constant names can be used in more than one nameset type. Conflicting names can be resolved using a type cast notation if necessary.

A nameset type definition may appear in or outside an object definition.

Nameset Values in a Variable or Array

One usage of nameset is to declare that a 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 **do** statement. The maximum value is 4,294,967,295 – the maximum for a **uint.**

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 named value. The function **isZero** can be used to test for zero.

For example:

```
# assigns unknown = 0, bicycle = 1, auto = 5, truck = 6:
nameset vehicles unknown, bicycle, auto=5, truck
# declare an array:
vehicles rigs[] # takes vehicle values only, with integer indexes
rigs = vehicles[auto,truck,truck,auto] # note - 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
DoorStats frontDoor
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

Another 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 bool 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 bool array with an index of nameset type to be compressed into a bit string, the maximum value in the nameset is 2047. All other types the maximum index is 4,294,967,295, the same as the maximum value of a **uint**.

Once declared, array members can be referred to by the nameset constant names as index.

Example:

```
    nameset TrainingType coder, site_design, graphics, fonts, writer, editor
    bool training[TrainingType]
    training[writer] = TRUE
    training[editor] = TRUE
.   training.site_design = TRUE# alternate notation
```

Code Statement

The code statement allows a string encoding to be defined. It defines a *code_name* with an associated character size and character values. Unlike **nameset**, it does not define a type. It is not a keyword and can be used for a name.

The format is:

code *code_name* [**.** *size*] *value_decl* […]

where *size* is the number of bits used for a character in this code. It must be from 1 to 32. If it is a power of 2 (1, 2, 4, 8, 16, 32) the characters are packed into a byte or unsigned integer. If not, there are unused bits. For example, character size 6 packs 5 characters into a **uint.32**, ignoring 2 bits. Size 5 packs 3 into **uint.16** wasting 1 bit.

Default size is 8.

The *value_decl* sets have the format:

start_value **(** *string_constant* **)**

where *start_value* is an unsigned integer constant.

Example of a code declaration:

```
code Octal.3 0('01234567')
```

and it can be used in a string declaration as:

```
string.Octal instr=Octal'0764'
```

with a period between string and Octal. Octal can store 21 character values in a uint.64 double-word.

A string constant in a code is identified by the code name prefixed to the constant.

Table: Standard Code Names

Code	Size	Values
ASCII	8	7 bit standard ASCII character values in 8-bit bytes
HEX	4	0('0123456789') 10('ABCDEF') 10('abcdef')
EBCD	8	IBM **EBCDIC** character code
DEC	4	Suitable for packed telephone numbers, packed decimal values, packed social security numbers, etc. 0('0123456789')
LATIN1	8	8-bit characters in the **ISO-8859-1** code
UTF8	8	UTF-8 characters (default)
UTF16	16	UTF-16 characters
UTF32	32	UTF-32 characters

Parameter Type Specification

A *param_type_spec* identifies a named set of types which can be sued to indicate which types are supported by a procedure. It is specified with the **partypes** keyword, for example:

```
partypes arithType: int float
```

This may be specified in an interface or object definition or as shared definitions. Additional param_types and their types allowed may appear after a comma.

The parameterized type name arithType represents a type which can be **int** or **float.**

The specified types allowed also can be an interface name or an asterisk. Specifying an asterisk means all basic types. A type preceded by the keyword **not** or **NOT** means it is not allowed. For example, the following *param_type_spec* allows any comparable type except **float**:

```
partypes key_type: _Comparable NOT float
```

Specifying **uint.*** or **int.*** means all sizes of **uint** or **int** types, respectively.

The *param_type_name* is available as a type name in procedure and other definitions in the object or interface or shared definition. Any of the eligible types for that name will qualify for that usage.

The *base_type* on a procedure definition can specify which parameter types are permitted by naming the *param_type_name* as the base type or as the type of a parameter or as the result type.

The standard library file **std/math.ss** defines several usable partypes names:

```
_NegMathType: int.* float
_PosMathType: uint.*
_MathType: int.* uint.* float
_IntType: int.* uint.*
```

Strings

A string is an object which resembles an array of characters. Strings are immutable. In SS each character internally is by default in the UTF-8 code. The 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 ("), possibly prefixed by a code name. 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), object names with simple members, variables and array elements with a simple index variable or integer constant can be embedded by enclosing them in curly braces if they can be evaluated to a string. 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 embedded inside a string constant which started with a quotation mark must have a preceding reverse slash.

A string constant can also be delimited by << and >>. This style can span multiple lines. Ends of lines except a line ending with << are included as '\n' inside the string constant except when << is followed only by a line end. Insertions are allowed as in a string constant which starts with a quotation mark.

Example:

```
string text = <<Line 1
this is line 2
this is line 3 with insert {abc[kk]}.>>
```

A string constant which begins with apostrophe or a quotation can be preceded by a code name. This does not apply to a string constant surrounded by << and >>.

String values are concatenated with a pair of vertical bars (||).

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 bytes.

A string can be indexed like an array (first index is 0) 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)
\u(dd), \u(dddd) or \u(dddddd) or \U(dd), \U(dddd) or \U(dddddd)	Unicode character; 2, 4 or 6 hex digits, leading 00 optional, in parentheses

There is no specific 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.

Regular Expression Pattern Strings

A regular expression pattern string is written with enclosing slash (/) marks, and it is scanned for correctness. Backslashes that aren't part of special character codes (like \n) will be preserved, rather than ignored or diagnosed, and change the meaning of the pattern. Some characters, such as question marks and plus signs, have special meanings in regular expressions and must be preceded by a backslash if they are meant to represent the character itself.

Curly braces in a pattern string do not allow insertions. The final slash has no folowing option letters.

Code Conversion in Strings

Converting a string in one code to another is easily done by simple assignment. Example:

```
string.UTF16 e_accent = UTF8'é'
string.LATIN1 e_tick
e_tick = e_accent # convert
```

A type cast can also convert a code:

```
string name = 'Jack'
string.UTF16 nickname
nickname = name.string.UTF16
```

String Expressions

Concatenation example:

```
string drink = 'tea'
string tasty = 'iced ' || drink # concatenated
```

A string acts like an 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** (counting from 0) 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. The index counts bytes, not Unicode "code points".

Assignment copies a value.

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

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

Concatenation is performed with the operator ||.

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

```
string a = 'abcdefghijkl'
print a.substr(2,5)  # prints "cdefg"
```

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.
byteSize	Returns bits per byte/character
capsCase	String returned has the first letter upper case, the rest lower, each word
clear	Reset string variable/array element to a zero length
codeName	Name (as a string) of the code sed
copy	Returns a copy of the string
count	Length of the string, the number of bytes
findLeft(string s)	Integer position of the first substring **s**, **null** if not found
findRight(string s)	Integer position of the rightmost substring **s**, or **null**
first(uint m)	Returns substring, the first **m** characters, **null** if none or if **m** is invalid
from(uint m)	Returns substring from character **m** up to the end, '' if none
last(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**
reverse	String with characters reversed. Invalid Unicode may result
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
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 accents) upper case
value(uint n)	Value of the n-th character/byte, as a **uint**

The built-in string function prototypes are defined in the standard library member **std/string.ss**. Note that copy, count and slice are also array functions.

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

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

Examples:

```
string s = 'abcdefghij', ss
int pos = s.findLeft('def')    # set to 4
ss = s.first(8)   # truncates to 8 characters
ss = s.from(2).before('h')   # gets 'cdefg' substring
string txt = 'apple, peanut, cheese'
# set array words 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	(base)	Absolute value
arccos	float	(base)	Arc Cosine
arcsin	float	(base)	Arc Sine
arctan	float	(base)	Arc Tangent
arctan(float **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
cos	float	(base)	Cosine
cosh	float	(base)	Hyperbolic Cosine
exp	float	(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	bool	Test for "Infinity" + or -
isNaN	float	bool	Test for "Not a Number"
isNeg	int float	bool	Returns true if < 0 else false
isNonZero	int float	bool	Returns true if <> 0 else false
isPos	int float	bool	Returns true if > 0 else false
isZero	int float	bool	Returns true if =? 0 else false
log	float	(base)	Logarithm, base **e**
log10	float	(base)	Logarithm, base **10**
maxValue(**y**)	float int	(base)	Maximum of base(**x**) and **y**
minValue(**y**)	float int	(base)	Minimum of base(**x**) and **y**
round(uint **y**)	float	(base)	Round **y** digits away from zero
roundEven(uint **y**)	float	(base)	Banker's rounding, even lowest digit
sin	float	(base)	Sine
sinh	float	(base)	Hyperbolic Sine
sqrt	float	(base)	Square root
tan	float	(base)	Tangent
tanh	float	(base)	Hyperbolic Tangent

The standard math functions are defined in standard library member **std/math.ss**.

Functions maxValue and minValue require the base and **y** to be the same type.

In addition, the identifier **_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. Also defined are **_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, if any, in parentheses. The object is an implicit argument to the function. A function may have no arguments and no parentheses.

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

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

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

Passing Arguments to Procedures

A procedure may be invoked upon an object. The object is then implicitly passed as a reference object.

A function or mode with no arguments, or with all arguments having defined default values, may be invoked with no parentheses.

A method never uses parentheses, even when arguments are passed.

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 object type definition.

- A **nameset** definition.

- A named constant declaration.

- A **partypes** declaration.

- An **interface** declaration.

- A procedure (**method**, **function**, **mode**) definition.

- A property declaration.

- An indexer declaration.

- A constructor declaration.

- A finalizer declaration.

- An **pragma** declaration.

Declaring Names

Variables and arrays can be declared:

- In a block, also a switch case block.

- In an object declaration.

- A variable with local scope can be declared in a **do** 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 an **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 bool, uint, int, float, string, any nameset type 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 type.

- 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 or outside an object declaration, 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 definition. 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 or a nameset type name.

The defined *name* by convention is 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:

```
public const uint 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 are allowed only in a procedure, including a method, mode, function, operator definition, indexer, type cast definition or finalizer.

Statements which control the flow of execution, meaning the progression from one statement to another, are called *control statements*. These are the **if**, **do**, **each**, and **switch** constructs, and subordinate control statements **return**, **break**, **repeat**, **raise**, **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***=**.

- 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 mode invocation can be a statement if it stands alone, not in an assignment or expression.

- A construct is considered to be an executable statement.

Control Statements

The four *constructs*, or multi-line statements, are **if**, **switch**, **do** and **each**.

The statements **exit**, **break**, **raise**, **continue** and **return** control the flow of execution and are also considered control statements.

If Construct

The *if_construct* has the following form:

```
if bool_expression
      block
[elsif
      block]...
[else
      block]
[when conditions
      block]
end if
```

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

Keywords **elsif, else** and **when** are not statements, but begin clauses in an **if** construct.

Multiple **elsif** blocks are allowed after **if** and before **else**.

Note that **else if** is invalid since nothing follows **else** on the same line.

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 [,...]
        block ...
[default
        block]
[when conditions
        block]
end switch
```

The expression on the **switch** must be an integer or string expression. The constants must all be the same type. 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 (the *case_block*) are executed. Execution goes to the **end** if it falls through the statements into a **case** or **default**, rather than fall through into the following case block. The **continue** statement overrides this, allowing fall-through. 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** before **case** or **default** overrides the implicit jump to the **end** .

The **continue** is allowed only as the last statement before **case** or **default** or **when** 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. 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 (which is a case block) indicates selection of any other unspecified value. It must follow all **case** statements.

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 switch
```

Final Block

The keyword **final** marks the start of a *final_block*. The purpose of the *final_block* is to allow actions at the end of each iteration of a **do** or **each** construct, such as incrementation.

Immediately following the *block* on a **do** or **each** construct, a **final** statement introduces a *final_block*, the (non-empty) block of statements after keyword **final**. This block of statements remains part of the block for name scoping. Execution flow falls into this *final_block*, through the **final** statement. Any incrementation or other action may be placed in the *final_block* if it is to be done at the end of each loop.

The *final_block* is executed on every repetition of the **do** or **each .**

A **repeat** statement in the block before **final** transfers to the *final_block*. A **repeat** is invalid in the *final_block*.

A **break** statement anywhere in the **do** or **each** construct exits the loop.

Do Construct

The *do_construct* is a block and is a loop statement. It has the form:

```
do [[type] assignment] [{while|until} test_expr]
        block
[final
        block]
[when conditions
        block]
end do
```

where *assignment* is executed once. 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 do statement and the *block, final block* and *when block*. If not typed, it must be a valid existing typed variable, and the value is retained after the end of the do loop for its normal scope.

The *test_expr* is a bool expression. The **while** or **until** test always takes place before the *block* on each iteration of a loop. If this is omitted, there is no loop, unless there is a **repeat** statement in the *block* or the *final block*.

Example of a **do**:

```
do int k = 1024 while k <> 0
        block
final
        k = k/2
end do
```

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

Another example:

```
do {while|until} test_expression
      block
end do
```

repeats the *block* as long as the *test_expression* remains **true** if **while,** or **false** if **until**.

Also,

```
do while true
      block
end do
```

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

Finally, a do with no **final** or **until** or **while**:

```
do
      block
end do
```

which executes the *block* without looping. This *block* may have declarations which are local in scope.

Each Construct

The **each** construct loops through an array or string or an object with certain capabilities, handling each value. There are four forms of the **each** construct:

```
[1]    each val [[k]] of array
            block
       [final
            block]
       [when conditions
            block]
       end each
```

which uses each element of the array, assigning the (local scope if not already typed) variable *val* the value, repeating the *block*. The type of *val* defaults to 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 size constant-valued index or a nameset index, it can be unrolled using a pragma.

Example of [1]:

```
float prices[string] = float['notebook'=2.55, 'pen'=0.77, 'paper'=1.98]
each cost[item] of prices
    printLn "item={item}, cost={cost}"
end each
```

A string can be used also, similar to an array. Variable *val* will be assigned each byte, left to right. Unicode characters are not identified. The keyword **ref** is not permitted.

```
[2]    each val [ [k] ] of string
            block
       [final
            block]
       [when conditions
            block]
       end each
```

which uses each byte of the string, the local scope variable **k** is assigned the index (a copy, not a reference) and implicitly local (or previously declared) variable **val** gets the value, type **string** and same code as *string*, repeating the *block*. The type of optional variable **k** is **uint**, and **k** will begin with 0. Variable **k** has localscope to the **each** block or it may be predeclared. The string cannot be altered in the block.

Variable **val** will be assigned each byte, left to right. Unicode characters are not identified.

```
[3]    each val of _Traversable_object
            block
       [final
            block]
       [when conditions
            block]
       end each
```

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

Finally, a generator function can be used:

```
[4]    each val of object.generator_function
           block
       [final
           block]
       [when conditions
           block]
       end each
```

which repeatedly calls the *generator_function* until it reaches **return final**.

Repeat Statement

The **repeat** statement is valid only inside a **do** or **each** block, and is not valid in the **final** block or **when** block portion. It causes execution to skip over statements to the **final** block or to the advance to the next array element. In other words, to the incrementation, the **final** block if it exists, or the next containing **end** at the block end on the nearest containing **do** or **each**.

Break Statement

The **break** statement is used to exit the nearest containing **do** or **each** block or **switch** block. It is typically used to exit a loop when a condition is reached. It is not used for a procedure body.

Example:

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

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 or the default.

This statement is not the same as **repeat**.

Exit Statement

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

The statement has an optional string parameter which is output with printLn before termination.

Return Statement

The keyword **return** is valid in an object definition. It can be used in a procedure body as a statement and in a property declaration.

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 or another object, named on the return. The return may name a reference to a new base.

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 has 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 implied value, or terminates after **return final** or execution falling through the end of the function with no return. A return with no value is allowed only when there is a default value variable.

Raise Statement

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

raise *condition*

where *condition* is a value of the type **_Condition**, which is a predefined object type, in **std/cond.ss**:

Nameset _ConditionLevel

A shared definition, in **std/cond.ss**. Used in defining the severity of conditions:

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

with meanings:

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

Object Type _Condition

A shared definition, in **std/cond.ss**. The object type _Condition is:

```
object _Condition
  static string[_ConditionLevel] _ConditionDefs = \
      [N='Normal', I='Information', W='Warning', E='Error', X='Failure']
  public _ConditionLevel level # a nameset
  public string msgText     # a message
end object
```

When Block

This block is executed only when a **raise** statement has been executed. Otherwise, it is skipped.

A *when_block* can appear at the end of a **do** or **each** block (after a final block), a procedure block, or a switch construct (just prior to **end**), but not part of **if**, **elsif** or **else** blocks, not a case block, and not prior to a final block. It is not part of an object definition.

It has the keyword **when**, followed by an optional list of allowed conditions, followed by statements and declarations on new lines. If there are no conditions, any condition is expected.

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 raised 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:

      raise _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
end if
```

Blocks

A block 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 implicit blocks, in the sense that they are groups of statements and declarations, and the declarations have scope. These blocks are:

- The *each block* or *do block* includes the entire **each** or **do** statement and the block executed. It also includes the *final_block* and *when_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. Declarations have scope in the switch block where they appear, including the when block if one is used.

- The *if_block* is all the statements from **if** to **end**.

- 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 procedure **end** statement. 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 after **final,** in the following block. It is not a true block because it is part of the same scope as the **do** block.

- The *when_block* is the statements from **when** through the following block to **end**. Its scope extends the preceding block. A *when_block* follows a **final** block in a **do** construct. It is permitted at the end of a **do**, **each**, **if** or **switch** construct and a procedure (**function**, **method**, **mode**) block.

Defining Objects

An *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:

```
object object_type [inherits object_type [(variant_name, ...)]] \
        [enables interface_name ,...]
    [partypes declarations]
    [variable and array declarations]
    [property declarations]
    [indexer declaration]
    [procedure definitions]
    [operator definitions]
    [type cast definitions]
    [abstract procedure/indexer/property definitions]
    [constructor definition]
    [finalizer definition]
end object
```

Object definitions cannot be nested.

An *interface_name_list* (separated by commas) appears after **enables**.

The **enables** and **inherits** phrases may appear in either order.

Variant names listed must exist. There must not be more than one named which are based on the same procedure.

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

Declarations

The allowed declarations in an object are:

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

Variable and Array Declarations

Variables and arrays are defined in an object in the form

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

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++) .

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. The name is public. 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.

Property Declarations

A property is a public name acting as a variable with controlled access, allowing read-only or write-only usage or requiring computation or procedure calls. There is no keyword "property".

The format is:

> [**abstract**] *type property_name* [*set_phrase*] [**,** …] [*return_phrase*]

where:

- In an *interface* or when the **abstract** keyword is used, any expression for **set** and the expression for **return** must be omitted. It is supplied when the definition is made later.

- 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 on an abstract property must be without action methods or expressions.

- A property declaration in an interface must be an abstract property.

- There is no access specified. It is always public.

- There is no initial value specified.

- *property_name* appears to be a public variable name. It acts like a function argument in the set phrases, and is not used in the return phrase.

- *set_phrase* is one or more of these:

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

    where at least one usage of the *property_name* is used as a value.

- Multiple set phrases are executed left to right.

- *return_phrase* is: **return** *expression.*

- 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 or use any accessible internal values.

- Property definitions can be intermixed with variable definitions which do not use public, private or final.

- A property value can be read-only (it has no **set**) or write-only, with no **return**.

- A property cannot be an array.

- A *property_name* cannot be static. It must be used as if it is a variable in an object, although it is not a variable.

Examples of Property Usage

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

```
object Dimens
   float wid, len # seen 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
end object

Dimens door
door.item_name = 'Rear entry'
door.width = 125 # centimeters
door.length = 300 # centimeters
printLn "{door.item_name} is {door.area} square cm in area."
```

Some usages for properties:

- Control or initialization, such as for a database.

- Simple functions with no arguments can be written as a return-only property. A function with no arguments is like a return-only property.

- Hiding of details, or encapsulation.

- Respelling or retyping.

- Simpler than functions and methods.

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]
```

The keyword **abstract** has the same meaning as on a property declaration or a procedure.

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 is shown by the index_type as the word **nameset** followed by an 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. It may instead return the result of a mode.

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
end object

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

Shared Definitions

A shared definition is defined only outside any object type definition; it is not an object definition.

Shared definition are outside an object definition. In addition to objects, they can be:

```
[partypes declarations]
[named constants]
[nameset types]
[interface definitions]
[methods based on null]
[functions and methods on a basic type or a nameset type or a partypes]
```

The names defined obey normal scope rules. They are global names, as if the shared definition is inherited by every object.

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

```
const float _PI 3.1415926535897932384626433832795 # ...
const float _E 2.7182818284590452353602874713526 6 # ...
const float _LN_10 2.3025850929940456840179914546 8 # ...
const float _ONE_DIV_LN_10 0.43429448190325182765 # ...
```

It is recommended that the names of constants be all upper case.

Methods defined which are not defined on a basic type have no base type; they are defined on **null**. These methods are not based upon any object.

Functions can be based on basic types or known nameset types.

Modes, indexers for an object, constructors and finalizers cannot be defined in the shared definition because they are not in an object definition.

Procedures are allowed to access previously defined constants, and they are allowed to use basic type or known nameset type definitions and interfaces.

There are no shared variables or arrays or properties.

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") implicitly supports this built-in property to test or inquire about the object's type:

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

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 or object constant. 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.

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

A bool 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.

Enhanced Type Objects

The inherited object type can be one of the basic types (int, uint, bool, string).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.

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

A procedure can be defined on an array of any type by specifying an object type followed by **[]** - a pair of square brackets. 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.

The *access* is one of the keywords **private** or **public**. If none is specified, the access is the default, protected, which is not specified with a keyword.

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 basic type name or **null**.

The base name can also be **parent** or the name of an object type inherited, to any level. These usages fulfill an **abstract** definition, or override a definition.

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 by placing its name in a parenthesized list after the object type name in the **inherits** phrase when a new object is specified. A property cannot have a variant.

Each variant is defined after the procedure block with the **variant** options keyword followed by a variant name. The variant name must be unique in the object.

The variant assumes the same base and the same procedure type. The arguments, if present, must match the same type and name as the base procedure, but default values can be different. If missing, the arguments and values are asumed to be the same. The result type must match if the variant is on a function. A result as an array can have different indexing.

Variants provide a way to avoid a need for abstract procedures and avoid a need to define additional object types.

A variant name procedure is not referenced by name anywhere in an object definition. It is not invoked outside the object either. Choosing a variant merely substitutes its body and arguments for the overridden procedure.

The variant block has shared scope with the procedure and the when blocks.

A procedure with a variant cannot be abstract.

Method

A *method is a* procedure with no return value. No return specification is provided. The arguments are not enclosed in parentheses. The form for a method definition is:

```
[procedure_options] [access] [base_type] method name [args_spec]
      block
[variant name
      block] ...
[when conditions
      block]
end method
```

Function

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

```
[procedure_options][access][base_type] function name[(args_spec)]->return_spec
 [repeat]
      block
[variant name
      block] ...
[when conditions
      block]
end function
```

where the option keyword **repeat** at the end implies this is a *generator_function*. The *args_spec* (including parentheses) is omitted if there are no arguments. The *return_spec* specifies the returned type and an optional local variable name and optional initial value constant-valued expression. It is required for a function.

Mode

A *mode* is a function with or without arguments, based on a defined object type, which implicitly returns a reference to the object. It may instead return a reference to a different *object_type*. The default result type is **self.** The characters **->** and a *result_type* can be omitted when the result type is **self** or the *base_type*.

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

```
[procedure_options] [access] [base_type] mode name [( args_spec )] [-> result_type]
      block
[variant name
      block] ...
[when conditions
      block]
end mode
```

Procedure Options

The *procedure_options* on the start of a procedure definition is any of the keywords **abstract** or **static**. These precede the *access* keyword, if it is supplied.

Abstract Procedures

If the *body* is omitted on a procedure or indexer, or a property definition has missing actions for **ref** or **set** in an object definition, it is an abstract 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.

The keyword abstract is not used on an object type definition, unlike other languages which require this when the "class" contains an abstract procedure.

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** in the body since there is no instance. It may only use or modify static variables. Static is not allowed on a mode.

The Function Return Specification

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

```
result_type [[return_item_name] [= init_return_value ]]
```

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

If the *return_item_name* is omitted but there is an *init_return_value* expression, the function block must be empty, and the function returns an "anonymous" value which is the expression. This is exactly the same as a return-only property. This can also be used in a type cast or operator definition.

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

When *base_type* is an object type, the procedure has access to internal data inside the definition and data shared from an inherited object type.

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 **final** as an indicator of failure, meaning it failed to create the object.

If the function defines a return variable with an initial value, the function will return that value if it has an empty procedure body.

Non_Object_Type Procedures

A non_object_type procedure is not based on a defined object type or nameset 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**. If a method is defined on base_type **null**, the invocation has no base.

If the base_type is not **null**, the value can be referenced in the procedure block as **this**.

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 as *access* on a procedure definition inside an object definition or interface. It means it will not be visible in inherited types.

Procedures in an object are defined with any appropriate access.

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.

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 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 arguments specification *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]

for each argument; 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 nameset type 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 make a copy then pass a reference to the copy.

Argument Passing

All value variables (types int, uint, float, bool) 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 which has no default value. The value passed is the value of the constant expression.

If all arguments have default values, an invocation can name any of the arguments in any order, specifying the name, an equal sign, and the value. This applies also to constructors.

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.

The statements after a **when** statement are a *when_block*, sharing scope.

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

Table: This, Self, Parent

this	The current instantiated object, used as a variable
self	The current type
parent	The base type of an inherited object type

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] mode default [( args_spec, … )]
        block
[when conditions
        block]
end mode
```

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 is invoked as a mode using the *type_name* as a way to create a new object.

There can also be one static constructor. It is invoked implicitly upon definition of the type, rather than instantiation of a variable. It has no arguments and it can only define static values. The static constructor cannot be explicitly called.

If there are no arguments on a constructor it will be invoked by default when an object is instantiated. If a constructor is not defined, default instantiation occurs.

The constructor does not have to be defined. A default constructor sets public values by naming them in the argument list. Unnamed public members are given default values.

A constructor with no arguments is called by the object type name.

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 or file.

A constructor must have statements in the block. The block (before **when**) cannot be empty.

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** after the keyword **mode**, in place of a mode name:

```
mode final
        block
[when conditions
        block]
end mode
```

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 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** after the *return_spec*.

It can be used only in an **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 the function end or **return final**.

A generator is a function, not a method or mode.

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 procedures (modes of functions) is called a *stack*.

If there is no base_type (it is defined 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. 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. It has no block, thus no local declarations.

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
getIndexType	property	String - the type name of the array's index
getName	property	String - nameset constant name for value in variable
getType	property	String - name of the type of the object
getValue	property	**uint** value of the nameset constant
getValuesArray	property	returns string-indexed array of values in the nameset type used as base
isNotNull	property	**true** if the ref to the object is NOT **null**
isNull	property	**true** if the ref to the object is **null**
testNull(val)	function	returns **val** if base is null, else returns the base; **val** must be the same type as the base

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

Type names are returned in the same case they are defined. In the case of a nameset, it returns "nameset typename", and a defined object as "object typename".

The properties getName, getType, getIndexType, getValue and getValuesArray are reflection properties. Reflection in SS is the ability to retrieve attributes of an object. Reflection is limited in SS to a these properties. If the array is fixed index, getIndexType returns 'fixed index'.

Function **testNull** is a null coalescing feature. One use is to provide a default value for missing array elements:

```
arr[i].testNull("Missing: arr[{i}]")
```

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 prefixing it with a selector period and the object name inherited or the word **parent**.

Example:

```
object ABC
    string x = 'Hello'
end object

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

# use it –
DEF  # prints: "Hello, World!" on instantiation
```

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

Defining Operators

Most of the operators can be defined for object types, in the object definition. 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 function with base type a single argument type, and result type shown, and in place of the function name the operator *op* is shown:

```
[base_type] function op [(type b)] -> return_type [var_name [= initial_value]]
        block
[when conditions
        block]
end function
```

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, a unary op, or a postfix op. The base *type* can be omitted or it can be **self**.

Equality (=?) is already defined to mean member-wise equality for objects, but can be redefined for any named object type.

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

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 the opposite pair is inferred.

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

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

Invalid operations on basic types cannot be defined. Postfix ++, for example, cannot be defined on bool or string.

Assignment cannot be defined.

Postfix or unary operators cannot be defined as binary operators and binary operators cannot be defined as unary or postfix.

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

The returned value is a reference for a defined object type, and 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 [var_name [= init_value ]]
        block
[when conditions
        block]
end function
```

The base type is the type in the left, the return_type is the type on the right, after the characters **->**. These types must be different. The keyword **self** means the base type of the enclosing object. The word **self** can then be omitted or the type name used instead.

There is no function name or operator.

Cast to bool is predefined for integer and float as the same as a nonzero test or the function **isNonZero**.

The *base_type* can be a basic type so long as the *return_type* is one which has no predefined definition. In other words, you cannot redefine casting an int to a string.

Interfaces

An *interface* is a declaration which defines the prototypes of a list of object functions, methods, modes, type casts, indexers, default named constants 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 definition may appear in or outside an object.

Default named constants are constant declarations which can be replaced, but are required. The default value is given.

An interface declaration has this format:

```
interface interface_name [enables interface_list]
    [partypes]
    [procedure prototypes]
    [property declarations]
    [type casts]
    [default named constants]
    [indexer declarations]
end interface
```

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.

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 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
end interface
```

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 -> num_type
end interface
```

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

```
object Obj enables Smaller
    num_type function Reduce -> num_type = this - 1
    end function
end object
```

Standard Interfaces

Special interface definitions are supplied in standard library **std/iface.ss**. 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) -> bool
end interface
```

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) -> bool
end interface
```

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 -> # ???? (TBD?)
    key_type Next -> # supply a 'next item' on the object
end interface
```

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

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

```
each item of obj
    block
end each
```

internally expands into:

```
do item = obj.First while item.Next.isNotNull
    block
final
    item .= Next
end do
```

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.

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 features to be implemented:

```
pragma use # bring in precompiled objects - NOT source code
         - not allowed inside objects, procedures, or any definition
pragma external [procedure] - using as a procedure body allows any code
pragma inline # makes this procedure expand inline, no procedure call used
pragma config
```

Certain behavior defaults for generated code or optimizations are controlled by pragma:

```
pragma option ...
pragma unroll
```

Example of **pragma external** -

```
float function sinh -> float
    pragma external sinh # assumes function, config names its location
    return sinh(this)
end function
```

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.

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 it.

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.

Printing Strings

The standard library member **std/std.ss** 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 of the output string.

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 defined as single argument methods on **null**.

Hello, World! Program

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

```
object @Main
   mode default
       printLn "Hello, World!"
   end mode
end object
```

It is automatically invoked by its constructor:

```
@Main
```

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/form.ss** defines an object type **_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 Indexes 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 its value.
```

Cookies

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

```
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 be used before any output is created.

HTML Headers

The same library member **std/std.ss** also defines a method **sendHeader** with a single string argument. The method is not based on an object. This must be used before any output is created.

Environment Strings

The library member **std/std.ss** also defines a function, based on a string, named **getEnv,** which returns the string value of the environment string name used as the base. Example:

```
string Host = 'HTTP_HOST'.getEnv
printLn "The host name is {Host}.<br>"
```

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 **std/directory.ss** 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)	bool	Deletes the directory and all of its contents
createDir(string path)	bool	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)	bool	Tests if a path is a directory
getPath(string path)	string	Returns full path name of a directory path
renameDir(string name)	bool	Rename/move the current directory

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

Failure when the result is not type bool 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 slash is always used.

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

These names can be obtained:

```
_Directory thisDir .= currentDir
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 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 **std/files.ss**.

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, 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	bool	Tests if a path is an existing file
isLegitName	string	bool	Tests if a path or file is a legal name
isWriteOK	string	bool	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 **std/io.ss**.

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

Table: Additional File Property

Function:	Base Type:	Result:	Description:
getLines	_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
string word = 'cloud'
do string line = inp.read
    if line.isNull  # stops on null return
        printLn "Word '{word}' not found."
        break
    end if
    if line.find(word) >= 0
        printLn "Word '{word}' found in line {line}."
        break
    end if
end do
```

Another Solution:

```
# this is an alternate solution to the problem - -
if 'data'.file.getLines.join('*').find('cloud') >= 0
    printLn "Word 'cloud' found"
else
    printLn "Word 'cloud' not found"
end if
```

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 **std/mysql.ss** 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 bool
close	_MySQL	Method, closes the database and clears connection data
isConnected	_MySQL	Property, returns bool
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 Indexes. 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, Indexes 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
values	string array
extras	string
quoted	bool - indicates whether values must be quoted

The values 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 **std/fmt.ss**. 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, ("Print" or "Produce") returns the produced string
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, ("minus") 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

All these modes return a _Fmt object reference. This object maintains the current settings and the current output string.

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

Default localization is in the **std/config.ss** file.

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 '_'	separate every 3 digits in integer part of float output if requested
decimal_point	'.' (default) or ','	decimal point in input/output

Regular Expressions

Regular expressions are implemented in SS using PCRE, "Perl Compatible Regular Expressions" and the functions and properties and the type **_Pattern** are found in std/regexp.ss.

Regular expression procedures operate on a string, either changing the string, or finding values in the string, or matching and validating the string. These use a pattern, a special string constant beginning and ending with a forward slash. The pattern string is the base, and other strings are arguments.

In order to do any regular expression procedure on a string, apply a function on a pattern string, For example, the function match returns true or false indicating the success or failure of a match.

Example:

```
if /def$/.match("abcdef")
    printLn "Found 'def' at the end."
if /^(Mon|Tues|Wednes|Thurs|Fri}Satur|Sun)day$/.match(day)
    printLn "Matched a day name: '{day}'"
end if
```

Table: Built-In Functions/Properties for Regular Expressions

Procedure	On:	Description
RegExpr(string *opts*)	String or pattern	Function, returns _Pattern type, saves optional *opts*.
match(string *str*)	_Pattern	Function, returns bool if pattern matches string *str*
replace(string str, string repl[])	_Pattern	Function, returns a string from scan of *str*, with matches replaced from array *repl*
split(string *str*)	_Pattern	Function, returns an array of values found in string *str* using separator defined by the pattern

Optimization of Regular Expressions

RegExpr "compiles" the pattern, saving an optimized form in the _Pattern 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:

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

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/form.ss**:

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

The complex type definition is in the standard library member **std/complex.ss** which implements the object data type **complex**, with associated operators and functions.

The following functions or properties are defined:

Table: Complex Mathematical Functions and Properties

Function/Property:	Result Type:	Returns:
abs	complex	Absolute value
arccos	complex	Arc Cosine
arg	complex	Argument, theta, angle for polar
arcsin	complex	Arc Sine
arctan	complex	Arc Tangent
conj	complex	Complex conjugate
cos	complex	Cosine
cosh	complex	Hyperbolic Cosine
exp	complex	Exponential, **e** to the power
Imaginary	float	Imaginary part
isNonZero	bool	Returns true if <> 0 else false
isZero	bool	Returns true if =? 0 else false
log	complex	Logarithm, base **e**
log10	complex	Logarithm, base **10**
Real	float	Real part
Reciprocal	complex	Complex 1/x
sin	complex	Sine
sinh	complex	Hyperbolic Sine
sqrt	complex	Square root
tan	complex	Tangent
tanh	complex	Hyperbolic Tangent
TimesI	complex	Multiply by the imaginary *i*

The property **isZero** is true if both real and imaginary parts are zero. Property **isNonZero** is true if either real or imaginary is nonzero.

A complex constant or value is written as complex(real part, imaginary part), using the constructor. The arguments are named and have default values of 0.0, so the constructor can be invoked using the argument names Real and Imaginary, as in this example:

```
complex ThreeTimesI = complex(Imaginary=3.0)
```

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
end interface
# 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.\""
   end method
end object
# animals Cat and Dog
Cat object inherits Animal # overloading Speak for Cat - - -
   method Speak
      printLn "{this.isType} {Name} says \"Meow!\""
   end method
end object

Dog object inherits Animal # overloading property Name for Dog
    Name set animal_name = Name return "Canine: " || animal_name.upper
end object

# 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
each who of pets
   who.Speak
end each
# expected output -
# Cat Fluffy says "Meow!",
# Cat Tom Cat says "Meow!"
# Canine: FIDO says "I am an animal."
```

Example - Binary Search Trees

Creating 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 enables _Traversable
  partypes key_type: _Comparable not float, data_type: *
  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
end interface
```

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