



The SS (Scripting System) Programming Language

Version: November 13, 2018

Table of Contents

Author and Copyright.....6
The SS Programming Language.....6
Features in SS.....7
Unusual Features.....8
Omitted Language Features.....9
Some Style Rules and Suggestions.....10
Libraries.....11
Terminology.....12
Reserved Words.....14
Table: Reserved Words.....14
Names.....14
Configuration.....15
Main Program.....15
Table: Standard Libraries.....15
Scope of Names.....16
Data Types Supported.....17
Integer.....17
Table: Integer Value Ranges.....17
Float.....18
Bool.....18
String.....18
Nameset.....19
Complex.....20
Decimal.....20
Value Types and Reference Types.....20
Expression Operators.....21
Table: Operators in Expressions.....21
Postfix Operators ++ and --.....22
The Construction Operator “!”.....23
Shortcut Operators AND and OR.....23
Table: AND and OR Operators.....23
Variables.....24
Implicit Type Conversions.....25
Integer Arithmetic.....25
Bool Expressions.....26
Null.....26
Type Casting.....27
Table: Legality of Non-String Type Casts.....27
Data Values.....28
Constants.....29
Arrays.....30
Array Elements.....32
Array Expressions.....33
Array Functions, Properties and Methods.....34
Table: Built-in Array Functions, Properties and Methods.....34
String as an Array.....34
Nameset Types.....35
Nameset Values in a Variable or Array.....35
Nameset as an Array Index Type.....36
Code Statement.....37
Table: Standard Code Names.....37
Strings.....38
Table: Escaped Characters in a String.....39
Code Conversion in Strings.....39
String Expressions.....40

Table: Built-in String Functions and Properties.....	41
Mathematical Functions and Properties.....	42
Table: Mathematical Functions and Properties.....	42
Procedures: Functions, Methods and Modes.....	43
Passing Arguments to Procedures.....	43
Defining Operators.....	44
Defining Type Casts.....	45
Declarations.....	46
Declaring Names.....	47
Named Constants.....	48
Statements.....	49
Control Statements.....	50
Single Statement Block.....	50
If Construct.....	50
Switch Construct.....	51
For Construct.....	52
Final Block.....	55
Repeat Statement.....	56
Break Statement.....	56
Continue Statement.....	56
Exit Statement.....	56
Return Statement.....	57
Raise Statement.....	58
Nameset _ConditionLevel.....	58
Object Type _Condition.....	58
When Block.....	59
Blocks.....	60
Objects.....	61
Variable and Array Declarations.....	62
Static Values.....	62
Static Procedures.....	62
Property Declarations.....	63
Examples of Property Usage.....	64
Indexers.....	65
Object Constants.....	66
Static Objects.....	67
Inheritance.....	68
Identifying the Base Object.....	69
Visibility of Names - Access.....	69
Enables Specification.....	70
Instantiation of an Object.....	70
Default Value at Instantiation.....	70
Object Member References.....	70
Type Switch Construct.....	71
Enhanced Object Types.....	72
Defining Procedures.....	73
Method.....	73
Function.....	73
Mode.....	73
The Function Return Specification.....	74
Non-Object Procedures.....	74
Procedure Syntax.....	75
The Base Type.....	75
The Arguments Specification.....	76
Argument Passing.....	76
The Procedure Body.....	77
Abstract Procedures.....	78

Variant Procedures.....	78
Constructors.....	79
Finalizer.....	80
Generator Functions.....	81
Invocation of a Procedure.....	82
Built-in Object Procedures.....	83
Table: Built-in Object Procedures and Properties.....	83
Name References in an Object.....	84
Interfaces.....	85
Interfaces and Procedures.....	87
Standard Interfaces.....	88
_Equality Interface.....	88
_Comparable Interface.....	88
_Traversable Interface.....	89
_Sortable Interface.....	89
Pragma Declaration.....	90
Member Access.....	91
Printing Strings.....	92
Hello, world! Program.....	92
HTML Form Data.....	93
Cookies.....	93
HTML Headers.....	93
Environment Strings.....	93
Directories.....	94
Table: Built-in Directory Functions and Properties.....	94
The File System.....	95
Table: Built-in File System Functions, Methods, Properties.....	95
Table: _FileAttribs Object Properties.....	95
File Operations.....	96
Table: File Operations Object Type Hierarchy.....	96
Open Files.....	97
Table: Interface _OpenFile Procedures.....	97
Table: Operations on an Open File.....	97
Table: Additional File Property.....	97
Example of Text File Reading.....	98
Another Solution.....	98
Date and Time Processing.....	99
Unfinished & Postponed Features.....	99
MySQL® Database Operations.....	100
Table: _MySQL Procedures.....	100
Object _MySQLtable.....	101
Table: Object _MySQLtable.....	101
Object _MySQLfield.....	101
Table: Object _MySQLfield.....	101
Formatting Strings.....	102
Table: Built-in Formatting Functions, Properties and Modes.....	102
Examples of formatting.....	103
Localization of Formatting.....	104
Table: Locale Options.....	104
Regular Expressions.....	105
Table: Built-In Procedures/Properties for Regular Expressions.....	105
Optimization of Regular Expressions.....	106
HTML and URL Functions.....	106
Table: HTML and URL Functions/Properties.....	106
Complex Data Type.....	107
Table: Complex Mathematical Functions and Properties.....	107
Example - Polymorphism.....	108

Extended Example - Binary Search Trees.....	109
Interface <code>_BST</code>	109
Object Type <code>'_TreeNode'</code>	110
<code>_TreeNode</code> Example.....	116
Object Type <code>_BalancedTree</code>	117
Factory Pattern Example.....	118
Language Design Considerations.....	119
Source Files.....	119
Everything Is an Object.....	119

Author and Copyright

Copyright © 2009-2018 - 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 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.

The **code** statement allows special codes to be used.

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.

Features in SS

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.

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

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

Names in SS are significant case words. The names ABC and Abc and abc are all different. There are a number of reserved words in SS. They cannot be used for names. The case of the reserved words is lower case.

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

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

The language is expandable. Types Complex and Decimal are defined object types, existing definitions provided. These can be expanded or replaced.

Unusual Features

- An array can have string indexes rather than integer indexes. Both fixed length and variable length arrays are supported. Arrays of arrays are possible. No two-dimensional “square” arrays are allowed.
- A number of array procedures are provided.
- 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 “==”. Semicolon is not used. These features help avoid some errors.
- The *mode* feature is different. It encourages an unusual style which avoids errors. A mode is a function based on a defined object type that implicitly returns a reference to its base object or a new object. It normally just sets an indicator or is part of the settings of an object.
- A generator function enables abstraction when obtaining a sequence of values.
- A list or tree can be made to operate like an array.
- SS has a parameterized interface and object type. Objects and functions can support multiple types.
- An enhanced object type can keep units apart, like Fahrenheit versus Celsius, or Meters versus Inches.
- The model for file manipulation is different, 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.
- 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 missing or simplified features:

- Different sizes (or precision) of floating point. One precision (double) is provided.
- Pointers and pointer arithmetic. These are omitted because of danger and misuse.
- Initialization is simplified.
- Garbage collection is used automatically.
- Structs. The object includes all struct capabilities.
- Control of namespaces; modules, packages, etc. Simple scope rules are used.
- Type punning, or union. Omitted because of danger and misuse.
- Events and event handling.
- Threading and parallel computation.
- Prefix ++ and -- are omitted (postfix ++ and -- are included), also the C language family ternary operators '?' and ':' which allow a choice in the middle of an expression are omitted.
- Direct input and output from a user. No graphic support. Programs run on a server.
- Closures, functions as arguments, callbacks.

Some Style Rules and Suggestions

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

```
default object @Main
{
    declarations and statements, including a constructor
}
```
- There are no executable statements, control constructs, or assignments outside of an object.
- There are no multiple statements on a line except where a control statement allows simple cases. This encourages easier modification.
- Comments follow a # sign, to the end of the line. The comment acts as an end of line.
- There are no multi-line or embedded comments.
- A statement can continue across lines if the last non-space character of a line or the last non-space character before a comment (#) is a reverse slash (\). Names and keywords, operators using two characters and numeric constants cannot be broken. The continuation mark reverse slash is not effective if it is in a comment or string constant.
- Multi-line string constants can be written using the << and >> symbols without using continuation marks. Embedded end of line breaks are value \n.
- The source file end of line is either \r\n or \n.
- Programs are expected to use indentation for readability. The “{” character starting a block is always on a separate line. (Exception for { }.)
- Readability suggestions –
 - surround assignment operators with blanks.
 - blank after a comma.
 - place “{” directly under the above statement start.
 - indent contents by at least 2 positions.
 - keep source lines short by using continuation.
 - 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)
 - \$(dollar sign)

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/** and assumed extension **.ss**. There are some standard library files which are automatically included, in directory **std/**. These define standard objects or procedures. 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 under **user/** are permitted.

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

Apache can be set to run any default file. The name **main.ss** is suggested.

Terminology

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

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

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

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

Reserved Words

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

Reserved words in lower case except and, or, not, null, false, true may be in upper case, as AND, OR, NOT, NULL, FALSE, TRUE. Other case forms are not reserved words.

Reserved words are also called keywords. There are 63 reserved words.

Table: Reserved Words

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

Names

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

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

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

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

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 an object type-named **@Main**, in effect, executing a line:

```
@Main!
```

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

Before that, all 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	
std/math	Mathematical constants and functions
std/mysql	MySQL functions, objects, properties
std/std	
std/string	String functions and properties

Scope of Names

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

All names are defined in an object. Those in a static object have global scope.

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

Data Types Supported

The standard data types in SS are integer, float, bool, nameset type names and string. They are also called basic types.

New types can be added as objects. Types complex and digital are available object definitions.

Integer

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

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

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

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 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 is **false**.

String

Strings are varying in length. There is no maximum length. The string type in SS is not terminated by a "null character" as in C/C++. Characters in a string are in the UTF-8 code, by default, using 8-bit bytes.

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. Strings are immutable; internal values cannot be modified.

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

Nameset

Nameset items are named constant unsigned integer values, grouped into a 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 with a variable or array using that nameset name as its type.

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 is found in the library **std/complex.ss**. It is NOT automatically included. Alternative implementations can be defined.

Decimal

Decimal type is available as a defined object type named Decimal. It is a very long signed integer. The current implementation allows up to 45 digits. Decimal variables are immutable; you cannot alter part of the value.

The decimal type is implemented by an object named Decimal which is found in the library **std/decimal.ss**. It is NOT automatically included. 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 types.

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
!	Construction operator	12	n/a	[special rules]
.	Member/procedure/type cast/code name selector	11	left	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 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 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 assignment	1	right	[special rules]
<>	Swap	none	n/a	(any, same)

Highest priority numbers are evaluated first.

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

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

```
abc += ijk = xyz = 123
```

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

```
a op= x
```

is the same as:

```
a = a op x
```

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

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

Examples:

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

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

The exponentiation operator ^ is right associative; the expression **a^{2³}** is the same as **a^(2³)**. 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, then increments it. For an object member, the left side then must be public or a property name with both **set** and **return**.

There are no prefix versions of these operators.

These can be statements because they alter a value.

The Construction Operator “!”

An exclamation mark ! can be used after a defined type name to create a new initialized object of that type, invoking the constructor and, if defined, the static constructor. The object created is returned. This operator cannot be used on the declaration of the item, nor on the left side of an assignment, nor on a basic type.

This cannot be used when the constructor requires parameters.

Shortcut Operators AND and OR

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

Table: AND and OR Operators

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

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

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

Variables

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

```
type name_list
```

Example:

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

It has a *type*, which is **int**, **uint**, **string**, **float**, 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, and is the same result used in the standard methods print and println, or the same as a value insertion in a string. Integers are exact, float numbers are approximate.

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

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

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

Integer Arithmetic

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

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

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

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 can be compared with `=?` or `\=`.

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.

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 not "type punning" which retypes without conversion. There is no type punning in SS.

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

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

#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**.

Data Values

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

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

Constants

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

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

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

A binary constant can be used for signed or unsigned integers. It has the form **0bddd...d** or **0Bddd...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.

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

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

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

Arrays

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

```
[ [index_type | nnn] ]
```

This is optionally followed by an initial value:

```
[ = array_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**.

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

An array of variables or array elements is allowed on the left side of an assignment. These variables are assigned left to right from elements of the right side expression, which must be an array. Missing values raise an error, extra values are ignored.

Example:

```
[a, b, c] = ['xyz', 'a123', "testing"]
```

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

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

Array Elements

Array elements are indicated as follows:

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

An array with nameset values as indexes 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= val #` applies the operator `op` with the scalar `val` to each element.
- `abc op def #` returns an array, applying the operator `op` on elements of the two arrays with matching indexes. If for one array a value is missing, the missing value takes a default of 0 or the empty string. Multiply (*) is not matrix multiply. The arrays are treated as vectors. The operator must be valid for the type of the array element values.
- Bit-wise operators can be applied to an array of integers.
- `abc op= def #` equivalent to: `abc = abc op def`, where `def` is an array or a scalar value.
- `abc .= proc #` where `proc` is a property or function, applies `proc` to each member of `abc`.
- `abc.proc #` also applies method `proc` to each member, or produces a new array if `proc` is a function invocation, applied to each member, or applies `proc` to the whole array if it is an array procedure.
- A procedure can be applied to an array. Examples:

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

Array Functions, Properties and Methods

Table: Built-in Array Functions, Properties and Methods

Name:	Returns:
clear	method, empty array. Delete element
count	the number of elements
firstIndex	lowest index in the array
flip	exchanges indexes and values of an array. Duplicates are lost
indexes	array of index values used for elements of an array
indexType	string - type name or 'fixed'
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 also
slice(uint m,uint n)	an array extracted from index m through n, m <= n
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 strings. The characters are indexed from 0 like an array. Indexing a string is not allowed in an Lvalue since strings are immutable.

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

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

Nameset Types

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

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

Nameset Values in a Variable or Array

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

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

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

For example:

```
# assigns unknown = 0, bicycle = 1, auto = 5, truck = 6:
nameset vehicles unknown, bicycle, auto=5, truck
# declare an array:
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

The second use for nameset allows the named constants to be indexes in an array with that nameset type as the index type. This provides an easy way to identify bits in a 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 index names.

Example, using one byte:

```
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 values
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
UTF16	16	UTF-16 characters
UTF32	32	UTF-32 characters

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), variables, array elements and even expressions that can evaluate to a string can be embedded by enclosing them in curly braces. Spaces in the insert are ignored. If a left curly brace is intended to be a character, precede it with a reverse slash.

Example:

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

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

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

Example:

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

If the << on the first line is enclosed in apostrophes, no insertion takes place. The ending line has no apostrophes.

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

String values are concatenated with a vertical bar. Concatenation is not performed inside a string constant except in an expression enclosed in curly braces.

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

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

A string can be indexed like an array (first index is 1) to retrieve a single character:

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

Table: Escaped Characters in a String

Escape Chars:	Meaning:
<code>\n</code>	End of line
<code>\r</code>	Carriage return
<code>\f</code>	Line feed character
<code>\t</code>	Tab character
<code>\\</code>	Reverse slash
<code>\{</code>	Left curly brace
<code>\xdd</code> or <code>\xdd</code>	Hexadecimal value dd (exactly 2 hex digits)
<code>\u(dd)</code> , <code>\u(dddd)</code> or <code>\u(ddddd)</code> or <code>\U(dd)</code> , <code>\U(dddd)</code> or <code>\U(ddddd)</code>	Unicode character; 2, 4 or 6 hex digits, leading zeroes optional, in parens

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.

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.
```

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* 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 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 = 'abcdefghijk1'  
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
byteValue	Value of the first byte, as a uint
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
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

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(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
Maximum(y)	float int	(base/y)	Maximum of base(x) and y
Minimum(y)	float int	(base/y)	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**.

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 *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 or with a pair of parentheses.

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

Defining Operators

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

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

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

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

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

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

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

The operators cannot be redefined for the basic types 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.

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, a value for a basic type.

Defining Type Casts

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

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

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

The return `type_name` cannot be `bool`. Cast to `bool` is predefined for integers 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 definition. In other words, you cannot redefine casting an `int` to a `string`.

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.
- An **interface** declaration.
- A procedure (**method**, **function**, **mode**) definition.
- A property declaration.
- An *indexer* property declaration.
- An **pragma** declaration.

Declaring Names

Variables and arrays can be declared:

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

```
int number
string name
```

- Arrays are declared with square brackets. All members must be the same type and all indexes either uint or int or string or a nameset 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 an object declaration, a static object, or any block. It has the following form:

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

The *access* keyword (**private** or **public**) is used only in an object type 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 which control the flow of execution, meaning the progression from one statement to another, are called *control statements*. These are the **if**, **for**, and **switch** constructs, and subordinate control statements **return**, **break**, **repeat**, **case**, **default**, **continue**, **else**, and **exit**.

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

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

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

A block delimits the scope of items declared in it.

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

Control Statements

Single Statement Block

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

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

If Construct

The *if_construct* has the following form:

```
if bool_expression block  
[ else { if_construct | block } ]...
```

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

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

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

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

Switch Construct

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

It has the following form:

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

The expression on the **switch** must be an integer or string expression. The constants must all be the same type. The 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**. This is not like the C or Java or similar languages.

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

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

Example of a switch:

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

For Construct

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

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

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

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

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

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

```
for int k = 1024 repeat k = k/2 while k != 0 block
```

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

And,

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

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

Also,

```
for block [final_block]
```

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

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

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

If the array has a fixed size constant-valued index, it can be unrolled using a pragma.

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.

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

which uses each element of the array, the local scope variable **k** is assigned the index and local scope variable **val** gets the value or reference if **ref**, repeating the *block*. The type of variable **k** is the index type, and **k** will begin with 0 if it is type integer. The type of **val** is the type of the array. The index **k** is given by value, not by reference.

If the array has a fixed index, the **for** can be unrolled by the compiler.

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.

Example:

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

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

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

An array and a string are traversable objects.

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

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

Final Block

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

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

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

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

The format of the combined for and final blocks is:

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

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

Repeat Statement

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

Break Statement

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

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

Example:

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

Continue Statement

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

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

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 before termination.

Return Statement

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

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

In a mode, it returns control with a result which is a reference to a base object or another object, named on the return..

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 terminates after a **return null** or a valueless **return**.

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 pre-defined object type, in **std/cond.ss**:

Nameset **_ConditionLevel**

Used in defining the severity of conditions:

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

with meanings:

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

Object Type **_Condition**

The object type **_Condition** has values:

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

There are standard conditions defined, including, for example these from **std/cond.ss**:

```
_CON_Overflow:      _Condition(level=E, message='overflow')
_CON_DivideByZero  _Condition(level=E, message='Divide by Zero')
_CON_ArrayBounds:  _Condition(level=A, message='ArrayBounds')
_CON_Failure:      _Condition(level=A, message='Failure')
_CON_Null:         _Condition(level=A, message='Null reference as object')
```

When Block

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

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

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

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

Blocks

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

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

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

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

Objects

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:

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

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

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

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

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

(ParName: int float)

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

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

Variant names listed must exist.

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

Object definitions can be nested.

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

Variable and Array Declarations

Variables and arrays are defined in the form

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

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

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

A variable definition example:

```
float rate = 0.035
```

Static Values

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

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

```
ABC.val
```

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

Static Procedures

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

Static is not allowed on a mode.

Property Declarations

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

The format is:

```
[abstract] type property_name [ set_phrase_list ] [ return_phrase ] [ , ... ]
```

where:

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

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

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

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

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

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

Examples of Property Usage

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

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

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

Some other 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 indistinguishable from a return-only property.
- Hiding of details, or encapsulation.
- Respelling or retyping.

Indexers

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

The form of an indexer is:

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

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

The *index_type* and *index_name* are used like a subscript. Index type is a basic type suitable for an array index. A fixed index is indicated by a *max_index_name* after the index name, following an equals sign. A nameset index 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.

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

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

Object Constants

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

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

Example:

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

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

Static Objects

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

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

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

It can have property variables.

The static object cannot have a constructor or finalizer.

It can internally define an object type, which is then available publically.

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

```
const float _PI 3.1415926535897932384626433832795 # ...
const float _E 2.71828182845904523536028747135266 # ...
const float _LN_10 2.30258509299404568401799145468 # ...
const float _ONE_DIV_LN_10 0.43429448190325182765 # ...
```

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

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

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

Modes and functions cannot be defined on the static object because it is not a new type and not an instance.

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

Inheritance

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

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

A variable V of a given object type T fits the words: V “is a” T .

If T inherits TT , then V “is a” TT also.

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

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

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

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

Identifying the Base Object

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

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

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

Visibility of Names - Access

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

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

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

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

Enables Specification

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

The *enables_spec* has the format:

```
enables interface_name,...
```

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

Instantiation of an Object

An object variable is a place-holder for a reference to an object. It is instantiated when it is assigned a reference to an instantiated object value 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.

Also, the construction operator (!) is used to create a new instance and perform its constructor.

Default Value at Instantiation

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

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

A 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.

Type Switch Construct

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

```
switch param_type_name
case type name | interface_name [,...]
    [ declarations ]
[ default
    [ declarations ]
]
end
```

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

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

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

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

Enhanced Object Types

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

The enhanced object is restricted in several ways:

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

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

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

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

However, multiplication and division 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 can be defined inside an object definition. Procedure definitions cannot be nested. They can also be based on a standard type.

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

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

On all procedure definitions, *access* is one of the keywords **private** or **public**. If none is specified, the access is the default, "protected." It also can be **ref** or **set**.

Method

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

```
[abstract | static] [access] base_type method name [args_spec]
{
  [statements and declarations] ...
}
```

Function

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

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

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

Mode

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

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

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

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:

```
return result_type [ return_item [ = init_return_value ] ]
```

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

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

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

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

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

If the function defines a return variable with an initial value or a construction mark, the function will return that value if it has an empty procedure body, a pair of curly braces.

Non-Object Procedures

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

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

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 on a procedure definition inside an object definition or interface. It means it will not be inherited.

Procedures in an object are defined with any appropriate access.

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

The Base Type

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

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

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

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

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

The Arguments Specification

The 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 ] [ , ... ]
```

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

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

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

Argument Passing

All value variables (types int, float, 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. The value passed is the value of the constant expression.

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

The Procedure Body

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

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

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

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

self means the current type.

parent means the base type of an inherited object type.

Abstract Procedures

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

Variant Procedures

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

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

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

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

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

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

Constructors

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

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

There is no *base_type* and the implicit mode name *type_name* is the name of the object type when the constructor is invoked.

The constructor 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.

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.

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 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** before the keyword **method**:

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

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

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

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

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

Generator Functions

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

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

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

A generator mode is not permitted.

Invocation of a Procedure

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

Invocation of a method will not use parentheses.

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

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

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

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

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

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

A property has no arguments.

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
indexType	property	String - the type name of the array's index
isNotNull	property	true if the ref to the object is NOT null
isNull	property	true if the ref to the object is null
isType	property	String - name of the type of the object
nullValue(val)	property	returns val if base is null, else returns the base; val must be the same type as the base
valuesArray	property	returns string-indexed array of values in a nameset

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

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

```
arr[i].nullValue('')
```

Name References in an Object

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

Example:

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

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

# use it -
DEF! # prints: "Hello, world!" on instantiation
```

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

Interfaces

An *interface* is a declaration which defines the prototypes of a list of object functions, methods, modes, type casts, indexers, 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.

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

An interface is not allowed inside an object definition.

An interface declaration has this format:

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

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

(ParName: int float)

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

A *type_name* which is to be excluded from an interface which has been named follows the interface name with the keyword **not** preceding. For example, the following *param_type_spec* allows any comparable type except **float**:

(key_type: _Comparable NOT float)

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

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

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

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

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

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

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

An interface that enables another interface inherits that interface definition, but it can overload any part of the enabled interface.

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

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

Example of an interface:

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

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

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

For example, the following interface:

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

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

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

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

Interfaces and Procedures

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

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

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

The order of the type names is not important.

Standard Interfaces

Special interface definitions are supplied in standard library **std/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) return bool
}
```

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

_Comparable Interface

The interface **_Comparable** is defined as:

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

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

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

_Traversable Interface

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

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

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

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

```
for each item = obj block
```

internally expands into:

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

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

_Sortable Interface

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

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

Basic type arrays are sortable.

Pragma Declaration

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

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

`pragma include ... #` bring in precompiled objects - NOT source code

`pragma external 'code file source' #` using this as a procedure body allows any code

`pragma inline #` makes this procedure expand inline, no procedure call used

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

`pragma option`

Member Access

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

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

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

The allowed declarations in an object are:

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

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

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

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

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

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

Printing Strings

The standard library member **std/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.

Example:

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

Note the string insertion in the message.

The method names `print` and `println` are not reserved words. They are in a static object as single argument methods.

Hello, World! Program

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

```
default object @Main  
{  
  default mode  
  {  
    println "Hello, world!"  
  }  
}
```

HTML Form Data

Values sent by an HTML form or by parameters attached to the URL are obtained through a single mechanism. The standard library member **std/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:

```
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.

Environment Strings

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

Directories

Handling directories (aka folders) is done through an object type **_Directory**. This object type defines some functions and methods to create and manipulate directories. The standard library member **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:
<code>attribs</code>	<code>_File</code>	<code>_FileAttribs</code>	Get the attributes of a file
<code>delete</code>	<code>_File</code>	(none)	Removes a file
<code>extension</code>	<code>_File</code>	string	Returns the extension
<code>file</code>	string	<code>_File</code>	<code>_File</code> object for a path or name
<code>isFile</code>	string	bool	Tests if a path is an existing file
<code>isLegitName</code>	string	bool	Tests if a path or file is a legal name
<code>isWriteOK</code>	string	bool	Tests if a file is writable
<code>name</code>	<code>_File</code>	string	The name part, with extension
<code>path</code>	<code>_File</code>	string	The full path for the file
<code>rename(string name)</code>	<code>_File</code>	<code>_File</code> or null	Change the file name
<code>move(string path)</code>	<code>_File</code>	<code>_File</code> 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:
<code>owner_uid</code>	return	string	Owner's user ID
<code>group_id</code>	return	int	Group ID (UNIX? LINUX?)
<code>node</code>	return	int	The file's Node (UNIX? LINUX?)
<code>permissions</code>	return/set	(??)	Permission bits (UNIX? LINUX?)
<code>size</code>	return	ulong	Length in bytes
<code>file_type</code>	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:
<code>_File</code>	<code>string.file</code>	object identifying the name/path <i>string</i>
<code>_TextFile</code>	<code>_File.text</code>	object operating on text lines in a file
<code>_ReadTextFile</code>	<code>_TextFile.input</code>	object reading text lines, file exists
<code>_WriteTextFile</code>	<code>_TextFile.output</code>	object writing text lines, replace a file
<code>_WriteTextFile</code>	<code>_TextFile.append</code>	object writing text lines, end of existing file
<code>_BinaryFile</code>	<code>_File.binary(object <i>blk</i>)</code>	object operating on a binary file of fixed sized blocks, the size of an object <i>blk</i>
<code>_ReadBinFile</code>	<code>_BinaryFile.input</code>	object reading binary blocks, existing file
<code>_WriteBinFile</code>	<code>_BinaryFile.output</code>	object writing binary blocks, replace/new file
<code>_UpdateBinFile</code>	<code>_BinaryFile.update</code>	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	property, returns a string (a line) or null
	rewind	mode, sets position to first line
_writeTextFile	write(string <i>str</i>)	function, writes <i>str</i> , returns bool
	writeLine(string <i>line</i>)	function, writes <i>line</i> with <code>\n</code> added, returns bool
	writeLine(string <i>lines</i> [])	function, writes all lines with <code>\n</code> added, returns bool
_ReadBinFile	rewind	mode, sets position to first byte (number 0)
	read(ulong <i>blknum</i>)	returns a string, one block, at data block number <i>blknum</i> , or null
_UpdateBinFile	read(ulong <i>blknum</i>)	returns a block, at block number <i>blknum</i> , or null
	write(ulong <i>blknum</i> , object <i>blk</i>)	function, writes a block, to block number <i>blknum</i> , returns bool
_WriteBinFile	write(struct <i>blk</i>)	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
for while (String line = inp.read) # stops on null return
{
    if line.find('cloud') >= 0
    {
        println "word 'cloud' found in line {line}."
        exit
    }
}
print "word 'cloud' not found."
```

Another Solution

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

Date and Time Processing

Unfinished & Postponed Features

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

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

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

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

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

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

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

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

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

MySQL® Database Operations

The standard library member `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:
<code>user(string uid)</code>	<code>_MySQL</code>	Mode, setting user id before connecting
<code>pwd(string pwd)</code>	<code>_MySQL</code>	Mode, setting password before connect
<code>defaultdb(string db)</code>	<code>_MySQL</code>	Mode, changing default database before connect
<code>connect</code>	<code>_MySQL</code>	Property, opens the database, returns bool
<code>close</code>	<code>_MySQL</code>	Method, closes the database and clears connection data
<code>isConnected</code>	<code>_MySQL</code>	Property, returns bool
<code>query(string sql)</code>	<code>_MySQL</code>	Function, returns object type <code>_MySQLresult</code>
<code>database_names</code>	<code>_MySQL</code>	Property, returns array of strings
<code>table_names</code>	<code>_MySQL</code>	Property, returns string array of tables in default db
<code>table_names(string db)</code>	<code>_MySQL</code>	Function, returns string array of tables in database db
<code>table(string tbl)</code>	<code>_MySQL</code>	Function, returns object type <code>_MySQLtable</code>
<code>fetch</code>	<code>_MySQLresult</code>	Property, returns an array with string index for a row
<code>fetchAll</code>	<code>_MySQLresult</code>	Property, returns an array of arrays (rows)
<code>count</code>	<code>_MySQLresult</code>	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:
<code>table</code>	string, name of the table
<code>num_fields</code>	number of fields
<code>fields</code>	array of objects of type <code>_MySQLfield</code> , Indexes are the string field names
<code>primaries</code>	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:
<code>name</code>	string, name of the field
<code>type</code>	string
<code>values</code>	string array
<code>extras</code>	string
<code>quoted</code>	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 <code>_Fmt</code> object
P	<code>_Fmt</code>	Property, returns the produced string ("Print" or "Produce")
X(int n)	<code>_Fmt</code>	mode, adds n blanks
W(int width)	<code>_Fmt</code>	mode, establishes width of next item
O	<code>_Fmt</code>	mode, resets all options to default
C	<code>_Fmt</code>	mode, shows thousands commas on N following
M	<code>_Fmt</code>	mode, shows negative numbers in parentheses, and
L	<code>_Fmt</code>	adds a blank on the right for plus values. Cancels Z
R	<code>_Fmt</code>	mode, sets left justification (default)
Z	<code>_Fmt</code>	mode, sets right justification
EOL	<code>_Fmt</code>	mode, sets leading zeros for N (int) item following, cancels M
D(int d)	<code>_Fmt</code>	mode, adds end-of-line characters
E	<code>_Fmt</code>	mode, sets d decimal places for numeric item following
S(string str)	<code>_Fmt</code>	mode, toggles mode for N (float) following to either display exponent style rather than fixed point or resume fixed
N(n)	<code>_Fmt</code>	mode, inserts the string value str, obeys W, R, L modes
		mode, inserts the numeric value n, obeys W, R, L, D, C, M, and Z modes. Ignores E mode if not float.
		Disallows M with Z mode.
		Float n ignores Z mode and C mode, and E mode implies a request for scientific mode

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

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

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

This formatting system is designed to optimize output and also enable compiler error checking. It is less error prone than systems based on C language `fprint`.

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

Examples of formatting

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

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


Regular Expressions

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

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

These use a pattern, a special string constant beginning and ending with a forward slash.

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."
```

Table: Built-In Procedures/Properties for Regular Expressions

Procedure	On:	Description
<code>match(string <i>str</i>)</code>	pattern	Property, returns bool if the pattern matches string <i>str</i>
<code>replace(string <i>str</i>, string <i>repl</i>)</code>	pattern	Function, returns a string from scan of <i>str</i> , with matches replaced from <i>repl</i>
<code>split(string <i>str</i>)</code>	pattern	Function, returns an array of values found in string <i>str</i> using separator defined by the pattern

Optimization of Regular Expressions

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

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

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

HTML and URL Functions

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

Table: HTML and URL Functions/Properties

Function/Property	On:	Description
<code>stripHTML</code>	string	Return string with all HTML or XML tags removed
<code>stripHTML(string array h)</code>	string	Return string with all HTML or XML tags removed, except keeping the tags in the array h , case independent
<code>HTMLspecials</code>	string	Return string changing apostrophe, quote, &, > and < to entities
<code>HTMLspecialsDecode</code>	string	Return string, with reverse of <code>HTMLspecials</code> conversion
<code>URLEncode</code>	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 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 $\neq 0$ 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).

Example - Polymorphism

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

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

```
interface Sayswhat # implement an interface - - -
{
  string Name set return
  method Speak
}
# the base object Animal - - -
Animal object enables Sayswhat
{
  string animal_name
  string Name set animal_name = Name return animal_name.title
  method Speak
  {
    println "{Name} says \"I am an animal.\""
  }
}
# animals Cat and Dog
Cat object inherits Animal # overloading speak for Cat - - -
{
  method Speak
  {
    println "{istype} {Name} says \"Meow!\""
  }
}
Dog object inherits Animal # overloading property Name for Dog
{
  Name set animal_name = Name return "Canine: " | animal_name.upper
}

# the program - - -
Cat Fluffy = Cat(Name='FLUFFY'), Tom = Cat(Name='tom cat')
Dog Fido = Dog(Name='Fido')
Animal pets[] = Animal[Fluffy, Tom, Fido] # an array
for each who = pets who.Speak
# expected output -
# Cat Fluffy says "Meow!",
# Cat Tom Cat says "Meow!"
# Canine: FIDO says "I am an animal."
```

Extended Example - Binary Search Trees

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

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

Interface `_BST`

The interface is parameterized, as follows:

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

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

Object Type '_TreeNode'

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

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

```
object _TreeNode (key_type: _Comparable not float, data_type: *) enables _BST
{
    self Left, Right
    key_type Key # kept in order on this member - protected access
    public data_type Data

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

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

# function which gives the number of nodes in the tree:
    public function count return int
    {
        if isNull return 0
        return Left.count + Right.count + 1
    }

# self Next return TBD
```

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

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

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

the indexer - -

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

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

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


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

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

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

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

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

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

This is a generator function.

It returns each tree key in key order, independent of key type, doing an In-Order walk

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

The SS Programming Language

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

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

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

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

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

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

The SS Programming Language

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

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

TreeNode Example

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

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

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

Object Type `_BalancedTree`

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

```
Object _BalancedTree (key_type: _Comparable not float, data_type: *) inherits

```

Factory Pattern Example

PHP 7.2	SS
<pre>// Factory and Car interfaces interface CarFactory { public function makeCar(); } interface Car { public function getType() : string; } // Implementations of the factory and car class SedanFactory implements CarFactory { public function makeCar() { return new Sedan(); } } class Sedan implements Car { public function getType() : string { return 'Sedan'; } } // Client \$factory = new SedanFactory(); \$car = \$factory->makeCar(); print \$car->getType() . "\r\n";</pre>	<pre># Factory interface interface CarFactory { public function makeCar() } # Implementations of the factory and car object SedanFactory implements CarFactory { public function makeCar() return Sedan c! { # returns c } } # no need for function getType or interface Car object Sedan # created by the factory { } # Client SedanFactory factory Sedan car = factory.makeCar() println car.isType # prints 'Sedan'</pre>

Language Design Considerations

Source Files

- Source is UTF-8 characters
- Multiple statements per line are avoided
- Curly braces are on separate lines
- Parentheses are not needed on control statements
- Confusion avoided on assignment and comparison equality operators

Everything Is an Object

- Enables a consistent functionality
- The mode procedure simplifies some design principles