The SS Programming Language

Version: June 15, 2019

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

Table of Contents

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

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

Copyright

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

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

Author

This programming language has been designed by John T. Bagwell Jr. of Sandpoint, Idaho. Bagwell had experience as a developer of compilers and supervisor of a team of compiler developers before retirement. He also published and presented a paper on code generation with local optimization in compilers.

The SS Programming Language

SS is a language designed for general application programming or server scripting. It is a complete programming language, not just for scripting, designed to be compiled. It is a pure object-oriented language, because all values are objects, 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.

If used as a scripting language, SS can be used in creating web sites and in creating tools to run under a server such as Apache. It can also be used for creation of stand-alone programs. A server environment simplifies input and output and simplifies user interaction. There is no standard graphical library.

Unlike many server-based programming languages, SS does not intermix source with HTML, and does not begin in HTML mode. There is no first line with "<?ss" in it.

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

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. A comparison is made for source file date being newer than library file date to invoke the compiler as needed.

Source File Format

Source files are text files in the UTF-8 code. Output strings to text files and database values and printed strings are by default in UTF-8 encoding. A UTF-8-encoded byte order mark (hex EF BB FF) is accepted and ignored if it is the first Unicode code point in the source text.

The end-of-line in the source is the Unix form using LF (linefeed) or the Windows form using the sequence CR LF (return followed by linefeed, 0x0C 0x0A). Either of these forms can be used equally, regardless of platform. In other words, CR is ignored before LF.

SS source files have the extension ".ss" after the file name.

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

Some of the Features of SS

- Every value is an object. Constants, arrays and expressions are objects. Procedures and names of types are not objects.
- All variables and arrays are declared and typed. It is a statically typed language, meaning variables and functions (etc.) are explicitly typed. It is a weak typing language, meaning there are implicit type conversion rules, not requiring type casting for built-in types.
- Limited inferred typing is used, for example in the **for** construct.
- Default access to object members is public.
- An interface feature is available. Procedure overloading and overriding are supported. Operator definition is included. It supports subtyping polymorphism. A procedure can have different implementations for different arguments, using parametric polymorphism.
- The case of all reserved words is lower case, except all upper case names are accepted for the six keywords AND, OR, NOT, NULL, TRUE or FALSE.
- A concern in the design is type safety. There should be fewer opportunities to confuse values, like string or array overflow, or types being misused.
- Another concern is maintainability. Making source edits easier is by design.
- Arrays and strings are designed to avoid overrun and bounds violations for safety and a degree of protection against hacking.
- An array can have string indexes rather than integer indexes. Arrays of arrays are possible. No "square" arrays are defined; there are no N-dimensional arrays.
- A number of array procedures are provided. Some array expressions are permitted, allowing optimized compiled code.
- The language is expandable. For example, type complex is a defined object type, an existing definition is provided. Array indexing and list manipulation are changeable.
- The **code** statement allows special "character" codes and character sizes to be used in strings.
- Identifiers can use additional European characters and accented letters as letters. This enables non-English words as identifiers.
- 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, …, PHP). There is less use of parentheses. Some operators are different; equality is "**=?**" rather than "**==**" and inequality is "<>" rather than "!=". Semicolon is not used to end statements.
- The word **new** is not used.
- SS is a late-binding language.

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

- The **mode** feature is an integral concept. It encourages an unusual style which avoids errors. A mode is a function based on a defined object type that implicitly returns a reference to its base object or a new object. It may just set an indicator or modify the settings of an object. It may return a new object reference. It is a mutator.
- A generator function enables obtaining a sequence of values.
- A user-defined list or tree can be made to operate like an array.
- Functions can support multiple types with a single source definition.
- A function or mode or method invocation with no arguments does not require a pair of parentheses.
- An enhanced object type can keep units apart, like Fahrenheit versus Celsius, or Meters versus Inches.
- The model for file manipulation is different, more object-oriented.
- Formatting for output is designed to be less error-prone, and is not patterned after Fortran.
- Regular expression functions have an optimization feature. Patterns can be prechecked.
- Values (variables, simple expressions, etc.) can be embedded in a string value, similar to PHP and PERL.
- Operators, array indexing and type casting can be defined.
- Variant procedures provide an alternative to abstract procedures in an object type definition.
- First class properties are supported.
- Some reflection features are included.
- A simpler status-handling **signal** and **when** are used instead of error handling in the **try**, **throw**, **catch** style.

Omitted Language Features

These are some of the omitted or simplified features:

- Different sizes (or precision) of floating point. One precision (double) is provided.
- Pointers and pointer arithmetic are omitted because of danger and misuse.
- Garbage collection is used automatically.
- Structs are not separated from objects.
- Type punning, or union, is omitted because of danger and misuse.
- Namespaces, modules, packages, etc. are not defined. Simple scope rules are used.
- Multiple statements per line are disalloowed.
- Events and event handling are left to library definitions.
- Threading and parallel computation are not implemented.
- Prefix ++ and -- are omitted; postfix ++ and -- are included.
- The C language family ternary operators '?' and '**:**' which allow a choice in the middle of an expression are omitted.
- Direct input from a user is omitted; there is no graphic support. Programs run on a server or as background tasks or use of an imported library or API.
- Closures, functions as arguments, callbacks, lambda expressions are not defined. Procedure variants supply some of these capabilities.

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

Source Style Rules and Suggestions

- A script consists of object definitions, interface definitions, code and typeset definitions, shared definitions and pragma statements.
- There are no executable statements, control constructs, or assignments outside of a procedure or a property.
- The main program is defined in the object named @Main, the constructor is the program.
- There are no multiple statements on a line except for simple if statements.
- Comments follow a # sign, to the end of the line. The comment mark acts as an end of line. There are no multi-line or embedded comments.
- A source line can continue across lines if the last non-space character of a line or the last non-space character before a comment (#) is a reverse slash (\). Names and keywords, operators using multiple characters and numeric constants cannot be broken. The reverse slash is not effective as a continuation mark if it is in a comment or string constant.
- A space or tab character (which is treated as a space) must be placed between source tokens if the meaning requires a space. For example, <~ is not the same as < ~.
- Multi-line string constants can be written using the << and >> symbols without using continuation marks. Embedded end of line breaks (except one appearing immediately after <<) have value \n (LF).
- Programs are expected to use indentation for readability. Readability suggestions -

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

- A numeric constant may use a single embedded underscore character between digits without affecting the value, for readability.
- SS language does not use these Unicode characters below U+0100, except in string or pattern values:

  ` grave accent
  ; semicolon
  × multiplication sign
  ÷ division sign
  all characters below 0x20 (blank) except HT (horizontal tab, \t), CR (\r) and LF (\n)
  all characters with value from 0xA0 through 0xBF

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

Libraries

The library inclusion statement format is:

pragma use *library_spec*

where *library_spec* is a specifier of a library and library member, as a name list of files with assumed directory **user/** with extension **.ss**. There are some standard library files which are automatically included, in directory **std/**. These define standard objects or procedures, interfaces and constants. The directory names are not reserved.

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

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

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

Standard libraries are implicitly included, as needed. They are (subject to change):

Table: Standard Libraries

Name (.ss):	Defines:
std/array	Array and list definitions
std/codes	Code names
std/directory	Directory functions, functions
std/files	File functions, functions
std/fmt	Formatting output
std/form	HTML and form input definitions
std/interfaces	Interfaces
std/io	File operations
std/math	Mathematical constants and functions
std/regexp	Regular expressions
std/status	Status
std/settings	Configuration file
std/std	Miscellaneous procedures etc.
std/string	String functions and functions
std/typesets	Typesets

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

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, terms which may need to be clarified are:

- ***object*** - the traditional object, an encapsulated entity containing data and associated functions and methods. In SS, every data item or value is an object, even the basic things like constants and variables of basic types. These implicit objects have predefined functions.
- ***object type*** - the new type which is defined by the construct with the word **object**. This is actually a "class" in common programming language terminology.
- ***enhanced object type*** - an object definition can inherit from a basic type, such as float, and enhance the meaning of that type. This allows a way to add attributes or restrict usage.
- ***basic type*** - the predefined types **int**, **uint**, **float**, **string** and **bool**.
- ***shared definition*** - the language assumes everything is an object, with some built-in functions and procedures. A **shared definition** is a way to add baseless methods, functions on basic types, nameset types, definitions of code names, object definitions, interfaces and named constants.
- ***array*** - a set of values, basic or object, all the same type, with a unique index per value.
- ***member*** - every item contained in an object is a *member* of that object. Also a single value in an array is an array member, or array element. Lists also have members.
- ***index*** - the subscript of an array. It can be an int, uint or string. The position identification for an array element.
- ***fixed-size array*** - an array with a predetermined number of contiguous members.
- ***list*** - a sequence of values of the same type, with no associated index.
- ***list value*** - an array value (a sequence of values enclosed in square brackets) preceded by a type name, a period, and the keyword **list**. Example: **uint.list[23,57,986]**
- ***statement*** - a line or construct which defines an action.
- ***construct*** - a multi-line statement.
- ***declaration*** - a line or construct which defines a type or defines a procedure. This is not an executable statement.
- ***block*** - a sequence of statements in a construct. A block acts as a statement.
- ***procedure*** - a method (a "subroutine") or function or mode, defined inside an object or on null or on a basic type. In some languages these concepts may be called methods or messages.

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

- ***method*** - a procedure or subroutine which does not return a value, thus it cannot be used as a primary element of an expression. It is used as a statement.
- ***function*** - a procedure which returns a value, usable in an expression.
- ***generator function*** - a function which behaves like an array in a for construct.
- ***null*** - a typeless constant which indicates no value, also non-existence.
- ***mode*** - a special kind of function. A type of mutator. 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 or many other uses.
- ***access*** or ***accessibility*** - 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 procedures and functions, not in inheriting objects. Defined with the option **private**. In a block, this access prevents visibility in a nested level.
  - ***protected*** - visible and accessible in the object and its procedures and functions, also in inheriting objects. Defined with the option **protected**.
  - ***public*** - default, not specified with a keyword. Accessible anywhere.
- ***indexer*** - a function which defines array indexing.
- ***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. These types are uint, int, bool, float and nameset.
- ***code*** - a character code for strings, specifying character size and values.
- ***nameset-indexed object*** - an object containing member variables (all the same type, all public) identified by the constant names in a nameset type. It resembles an array but uses object member accessing rather than subscripting. It has no inheritance and no procedures.
- ***property*** - a pseudo-variable, which may be read-only or write-only or both read and write.

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

Reserved Words

Reserved words cannot be used to define names for a program; they have predefined uses in the SS language. Reserved words are also called keywords.

Reserved words are in lower case, except the six words **and**, **or**, **not**, **true**, **false** and **null** may be may also be in upper case for readability. For example, the word **null** is also recognized if it is **NULL**. Other case forms of these keywords are not reserved words; they should be avoided.

There are 64 reserved words, including the six alternative forms.

Table: Reserved Words

$	case	final	interface	null	protected	signal	uint
and	code	float	list	NULL	public	static	ulong
AND	const	for	long	object	ref	string	until
begin	continue	funct	method	of	repeat	switch	uses
bit	else	if	mode	or	return	then	ushort
bool	enables	implicit	nameset	OR	self	true	variant
break	false	inherits	not	pragma	set	TRUE	when
byte	FALSE	int	NOT	private	short	typeset	while

Names

A name, or identifier, must begin with a letter. Digits or letters (in either case) are allowed after the initial letter. Names are unlimited in length. The case of letters is significant. Spaces may not be embedded.

Also considered as a "letter" for use in a name are the characters dollar sign (**$**), underscore (**_**), at-sign (**@**) and exclamation mark (**!**). An initial underscore and initial at-sign are used for language-defined names; user-defined names should avoid initial underscores and at-signs.

The single dollar sign (**$**) is a reserved word, used in a procedure or function, meaning the current object's value. It is the same as the word 'this' in some programming languages.

The single exclamation mark (**!**) is a supplied mathematical function, computing a factorial.

The single at-sign (**@**) is a method based on null, with a string argument. It outputs the string with end-of-line characters. It is an "echo" or "print" method, customarily pronounced "out".

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

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

Legal names: inTheMiddle, a1, @place, time_worked, arrêté!, days@work, $cost.
Invalid names: 1A, two words, A&B, x-hyphen, ¥999, 2×4.

Main Program

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

```
@Main
```

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

Settings

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

```
pragma settings
```

followed by *command* lines of this form:

```
command value1 value2 …
```

For example, it contains these lines, defining the format of displayed numbers:

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

A modified settings file is permitted. It will contain overrides to the standard settings.

Blocks and the Scope of Names

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

A name has a scope limited to the object it is in when **private** is specified, or to the *block* in which it is defined. The scope begins where the name is defined. At the end of that block, the scope is terminated and the variable, array, string, or object is erased and not visible.

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

All names defined outside an object have global scope, available anywhere. Data (variables and arrays) are not allowed outside of an object.

Names of object types, nameset types, typeset names and interface names are in the same name category and a name must be unique for those usages.

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

Data Types Supported

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

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

Integer

Integers have no fractional part, and are (by default) signed 4-byte integers, with type keyword **int**. Integers of other sizes can be declared. They are declared as **int.64** (**long**), **int.32** (**int**), and **int.16** (**short**). There are no **int.1** or **int.8** types. All of these are signed. The appended number is the number of bits used.

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

Alternate names can be used as shown. 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	byte	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.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

(The definition of this type can be changed if hardware does not support the IEEE standard.)

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.

There are no constants denoting the IEEE-754 negative zero, infinity, and not-a-number (NaN) values for the float type. These may have multiple internal values.

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**.

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

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. True has value 1 and false has value 0 internally.

The **if** statement and other places require a bool expression. The basic arithmetic types may be type cast to **bool**. A nonzero value becomes **true**, a zero or **null** value becomes **false**.

String

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

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

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

There is no separate type for single characters.

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

Nameset

Nameset items are named constant unsigned integer values, grouped into a new type name. This is similar to the C language 'enum', but more restrictive because the names are typed. A nameset constant can be used only as a value in a variable or array using the correct nameset type name as its type, or as a member name in a nameset-indexed object with the same nameset 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 definition is found as an object type in the library **std/complex.ss**. It is NOT automatically included. The type name **complex** is not a reserved word.

Alternative implementations can be defined.

Value Types and Reference Types

Value types are simpler data types which pass a copy of a value to a procedure, and which copy a value when they are assigned. These are the builtin or basic types except string, and nameset typed values.

Reference types are more complicated. When reference types are passed to a procedure, a copy of a reference (a hidden pointer) is passed. When an assignment is done, the reference is used, rather than the value. These are arrays, strings, nameset-indexed objects 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 specific copy or reference behavior on their arguments.

In an object definition, a name of a reference type item is always treated as a reference.

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

Objects and Inheritance

Every value is an object, and has an object type.

Objects support single inheritance, and they also support multiple interfaces.

There are some implicit "inheritances" which are implied by usage. These also imply standard interfaces, described in a section below. These implicit object types and interfaces can be redefined or modified.

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

Expression Operators

Table: Operators in Expressions

Operator	Description	Priority	Associativity	Operand Types
.	Member/procedure/type cast/code name selector and more	11	right	n/a
++	Postfix incrementation	10	n/a	int
--	Postfix decrementation	10	n/a	int
+	Unary +	9	n/a	int float
-	Unary -	9	n/a	int float
~	Unary bitwise complement	9	n/a	uint bool.nameset
^	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	uint bool.nameset
<~	Bitwise left circular shift	7	left	uint
~>	Bitwise right circular shift	7	left	uint
<*	Bitwise left shift	7	left	uint
***>**	Bitwise right shift	7	left	uint
+	Addition	6	left	int float
-	Subtraction	6	left	int float
||	Concatenation of strings	6	left	string
|	Bitwise inclusive or	6	left	uint bool.nameset
~	Bitwise exclusive or	6	left	uint bool.nameset
=?	Equality comparison	5	left	(any)
<>	Not equal	5	left	(any)
<	Less than	5	left	int float string
<=	Less or equal	5	left	int float string
>	Greater than	5	left	int float string
>=	Greater than or equal	5	left	int float string
not, **NOT**	Logical NOT (unary prefix)	4	n/a	bool
and, **AND**	Logical AND	3	left	bool
or, **OR**	Logical OR	2	left	bool
=	Assignment	1	right	[special rules]
***op*=**	Same as left = left *op* right	1	right	[special rules]
.=	Function/mode assignment	1	right	[special rules]

Highest priority numbers are evaluated first.

Assignments do not produce a value.

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

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

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

abc += ijk = xyz .= abs

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

a *op*= x

is the same as:

a = a *op* x

where **a** must be an Lvalue or an array.

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

Examples:

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

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

The 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.

Division of an integer by an integer produces a truncated (toward zero) integer result.

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.

Postfix Operators ++ and --

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

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

There are no prefix versions of these operators.

Unary Operators

The unary operators **+**, **-**, and **~** are restricted in placement. They may not follow a binary or unary arithmetic operator or a relational (comparison) without a space. They may follow a left parenthesis, a left square bracket, an assignment operator, a comma, or the logical operators **and**, **or**, **not**.

Shortcut Operators AND and OR

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

Table: AND and OR Operators

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

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

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

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

Variables

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

type name_list

Example:

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

It has a *type*, which is **bool**, **int**, **uint**, **string**, **float**, or a nameset type name, a nameset-indexed object type or an object type name.

An initial value can be assigned in the declaration:

int maxSize = 1000

Array declarations can be mixed with variable declarations:

float xyz, totals[], data[string]

List declarations are also allowed, mixed with variables and arrays:

string name1, name_array[100], list titles = string.list['duke','baron','earl','prince']

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

int ijk, string str[]

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

Implicit and Explicit 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 or loss of significant digits in converting from float to int is ignored.

A bool value can be explicitly type 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 to type bool requires an explicit type cast.

Conversion from a numeric to string is allowed by type casting, and is the same result used in the standard methods **@** and **@noEOL**, or the same as a value insertion in a string. Integers are exact, float numbers are approximate.

Conversion from a string to a numeric value may cause an error if there is no valid conversion after trimming off blanks. An empty string and a value too large to be an int both signal a conversion status.

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

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

Integer Arithmetic

Integers with widths 32, 16, or 8 bits or 1 bit (unsigned only) "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 (truncation) can be specified by type cast; abc**.**int**.**16 (or abc.short) extracts the least significant 16 bits of the integer abc.

An integer divided by an integer truncates toward zero. It does not yield a float result.

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

Bool Expressions

A **bool** expression has a value of **true** or **false**. These are also reserved words for the values. Arithmetic on these values is disallowed; the value of **true** is nonzero but not specified. A non-bool value cannot be compared with **true** or **false**. Bool values cannot be compared with =? or <>. The same result is obtained by use of **and**.

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

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

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

One example:

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

Null

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

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

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

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 to:	uint	int	float	bool
uint	Yes	#1	Yes	#2
int	#1	Yes	Yes	#2
float	#3	#3	Yes	#2
bool	#4	#4	No	Yes

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

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

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

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

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

Data Values

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

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

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

Constants

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

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

There is no support for octal values. A leading zero is ignored in a numeric constant.

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

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

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

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

Any numeric constant (except hexadecimal) may have a single underscore character embedded between digits for readability, without affecting the value. This may help in counting digits for long values.

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

Arrays

An array is declared with one or more *array_spec* definitions, as follows, after the type then the *array_name*:

[[*index_type*] **] ...**

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

The *index_type* can be **int** (any size) or **uint** (any size except **uint.1** or **bit**) or **string**. The default *index_type* is **uint** (**uint.32**). An array index cannot be type **float**, **bool** or an object type or nameset type. A *typeset_name* can be used for the *index_type*.

If a positive nonzero integer constant is given rather than a type name between the square brackets, the array is fixed-length, with **uint** indexes of value 0 up to but less than the number indicated. In other words, the declaration:

```
string Names[80]
```

defines an array, N**ames**, of 80 strings, with indexes 0 through 79.

The *array_spec* is optionally followed by an initial value:

[**=** *array_constant*]

The *type_name* may be omitted and implied on this constant to be the same as the array name's type.

There are no multiple dimensional, or rectangular arrays. An array of arrays is permitted, by repeating the *array_spec* part, in square brackets, after the *name*. An array of arrays is known as a *jagged* array because the array lengths can vary. Multiple levels are allowed.

When the index is **uint** or defaulted, the minimum index is 0 for the first element. The minimum is unspecified for a (signed) **int** index; it can be negative. The maximum index or maximum number of elements of an array is unspecified.

Examples of array declarations:

```
float scores[ushort]
int cards[52]
float table[][string] # an array of string-indexed arrays
string arr[]  = ['a','1','last']
float cost[string]
```

Initially an array which is declared as shown, if not fixed-length, has no elements. An empty array value is written as *type*[]. Values can be assigned to single or multiple elements as shown:

```
int arr[] = int[100, 25, -6, 94]
arr[6] = 'XYZ'      # skipping indexes 4 and 5, remaining unassigned
```

The index values in the initial value can be omitted or specified before the values, with a colon:

```
int arr[] = [23,-88,10:123,345]  # defines 4 array elements, indexes 0,1,10,11
```

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

```
float salary[string] = ['driver':12_000.00, 'sales':15_000.00]
```

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

```
salary['CEO'] = 249_995.00  # adds a new element, CEO's salary is $249,995.00
```

An array of arrays can have an initial value:

```
string classroll[string][string] = ['Lit':['Walt Whitfolk','Will Shakefist'], \
      'Geometry':['Archie Medes','Ima Euclidian','Neva Cross'], \
      'Gym':['Wilt Nicklaws']]
```

For an array of arrays, a reference with fewer indexes is a reference to an array.

An array is not instantiated unless it is assigned an initial value or an array value is assigned or a reference to another array is assigned. Declaring an array establishes a place-holder for a reference to an array.

A reference like `Arr[]` as a left value (an Lvalue) in an assignment, where `Arr` has uint or int indexes and no index is given, is assumed to create an index value which is 1 higher than the maximum already used index value, or 0 if the array `Arr` is empty. This is invalid for string indexes and fixed-length arrays.

A missing array element reference for a non-fixed-length array returns **null**. Existence can be checked with **isNull** or **isNotNull** or **testNull**.

A fixed-length array has all elements present; none are missing. Unassigned values are zero-valued, or false if bool, or empty strings if the array type is string. Unassigned nameset values are zero, even if no name has a zero value. Unassigned objects are **null**.

By default, an array which is not fixed-length is implemented as a 'map', a tree structure, using an object type described in an Appendix. The indexes are maintained in sorted order. The storage and indexing technique for a specific object type can be defined by use of an indexer.

Fixed-length arrays are pre-allocated to the defined number of elements, as a contiguous set of values or references/pointers.

Standard Definitions Used for Arrays

An array and array elements implicitly enable these interfaces and associated objects:

Interfaces:

_Array **_Traversable** **_BackTraversable** **_Comparable** **_BinarySearchTree**

Objects:

_Node **_ArrayRoot**

Also, several typeset names are used:

Typesets:

_KeyType **_IntType** **_DataType**

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

Array Expressions

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

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

- abc = xyz ◄ both are arrays of the same type; copies the reference. Both arrays refer to the same data.
- abc = xyz.Copy ◄ assigns a reference to a new copy of xyz. They refer to different data sets.
- abc = xyz ◄ where the types differ, but can be converted, builds a new array and assigns a reference to it.
- abc =? xyz ◄ is true if the two arrays have the same data and dimensionality and indexes. Inequality also is allowed.
- abc *op*= xyz ◄ is equivalent to: abc = abc *op* xyz, where xyz is an array or a scalar value.
- abc *op*= *val* ◄ applies the operator *op* with the scalar (non-array) *val* to each element.
- abc *op* xyz ◄ 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, it is element-by-element multiply. The operator must be valid for the type of the array element values.
- abc **.**= *proc* ◄ where *proc* is a function or procedure (function, method, mode), applies *proc* to each member of abc.
- abc = *genfunc* ◄ where *genfunc* is a generator function invocation, sets abc with an array of the values and indexes returned from *genfunc*.
- abc.proc ◄ also applies method proc to each member, or applies proc to the whole array if it is an array procedure.
- Bit-wise operators can be applied to an array of unsigned integers, or to bool**.**nameset.

Examples:

```
int cnt = iarr.count   # gets the number of elements in array iarr
arr .= sort            # sorts the values in an array with integer indexes
arr.sort                # another sort expression
a_array.Clear          # removes all elements
arr[4].Clear            # removes the 5th member of arr
arr += 2                # add 2 to each element
```

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

Array Values and Array Constants

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

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

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

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

Example, initializing in a declaration:

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

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

If desired, the type of the array can be placed just before the left bracket. The index types must all be the same and the index type cannot be specified in the constant.

An array value can appear in an array expression, or an array assignment on the right. The type name must appear before the left square bracket unless it can be determined from context.

Assignment to an Array Value

An array value can appear on the left of an assignment if it follows these rules:

- The number of members matches the array on the right side, or the right side is a single value.
- All members of the left side are Lvalues, variables or array elements, and they are all the same type as the array on the right side.
- A type name does not appear before either side's left square bracket. It is inferred from the right side or from the type of the first element of the left.
- The right side is evaluated fully before assignment is done element-by-element left to right.
- Neither left nor right side actually creates an array.

This left side array value feature enables useful capabilities:

```
[a, b] = [b, a] # swap a with b
[d, a, b, c] = [a, b, c, d] # rotate the four values right circularly
[a, b, c] += 100 # add 100 to 3 variables
[aarr[], barr[], carr[]] += ['a44','z01','mn5-2'] # push values onto 3 arrays
```

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

Table: Built-in Array Functions and Methods

Name:	Returns:
Clear	method, empty array, delete array element
Count	the number of elements
Copy	Copy the array
Current	current element (a reference; usable as Lvalue)
Each	generator function; sequences through array
First	index of the element with the lowest index; sets Current
Flip	exchanges indexes and values of an array. Duplicates are lost
getIndexType	string function, type name of the index
Indexes	array of index values used for elements of an array
Join(string sep)	string, elements as strings, concatenated with separators sep
Last	index of the element with the highest index; sets Current
MaxValue	maximum value in the array
MinValue	minimum value in the array
NewIndexes	array of values in index order; new ulong indexed result
Next	mode; moves Current, returns reference
Pop	return the last element value of an array and remove it
Prev	mode; moves Current, returns reference
Push(val)	adds new last elements on an array with integer indexes, type of val is same as base type, val is array, variable or list
Reverse	a new array, with ulong indexes, in reverse order by index
Slice(ulong m,n)	an array extracted from index **m** through **n**, **m** <= **n**
Sort	a new array, ulong indexes, sorted
Unpack	array created from a name-indexed object, string indexes from nameset
Value	value of an element

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 bytes, or a character of an appropriate size for the code used. The bytes are indexed from 0 like an array.

Indexing a string in an Lvalue is not allowed since strings are immutable.

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

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

A **for** construct treats a string as characters, not bytes.

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

Lists

A list is not an array. It has no index for each value. The individual values cannot be altered, so it is partially immutable. In some ways it is more like a string.

A list cannot be embedded in a string.

A list is implemented by a single-linked list.

A list can be checked with functions **Count** or **isNull** or **isNotNull** or **testNull** to see if if has no values.

The assignment array = list or list = array converts a list to or from an array with default uint indexing.

Table: Built-in List Functions and Methods

Name:	Returns:
Clear	method, empty the list
Copy	copy the list
Count	the number of elements
First	return the first element value
Join(string sep)	string, elements as strings, concatenated with separators sep
PopFirst	return the first element value and remove it
PushFirst(val)	add a new first element

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

Standard Definitions Used for Lists

A list implicitly enables these interfaces and associated objects:

Interfaces:

_List **_Traversable** **_Comparable**

Objects:

@ListNode **@ListRoot**

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

Nameset Types

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

```
nameset nameset_name name [:unsigned_integer_value] [, ...]
```

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

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

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

Nameset constant names can be used in more than one nameset type.

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

Nameset value names are not permitted as array indexes.

Each nameset declaration implicitly creates an array which resembles a static array. The array for nameset NSet (for example) is named **Values** which has **string** indexes and **uint** values, addressed as **Nset.Values**.

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

Nameset Values in a Variable or Array

One usage of nameset is to declare that a variable can be assigned values from the list. The variable can then be tested for equality with a name from the nameset, or the names can be used in a **switch** or a **do** statement. The maximum value is 4_294_967_295, which is the maximum for a **uint.**

Items in a nameset variable cannot be compared to integer values. An empty or uninitialized nameset data value has a value of 0, which may or may not match a named value. The function **isZero** can be used to test for zero.

For example:

```
# assigns unknown = 0, bicycle = 1, auto = 5, truck = 6:
nameset vehicles unknown, bicycle, auto:5, truck
# declare an array:
vehicles rigs[] # takes vehicle values only, with integer indexes
rigs = vehicles[auto,truck,truck,auto] # assign 4 values
rigs[2] = bicycle # change the third value
# another example - -
nameset Tstat closed, reading:8, writing, open:15 # values are 0, 8, 9, 15
nameset DoorStats open, ajar, closed # note the apparent name conflicts
DoorStats frontDoor
frontDoor = ajar
@ vehicles.Values['truck'] # prints: 6
```

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

Nameset-Indexed Objects

A nameset-indexed object is a variable which is an object whose members are all the same type and the member names are the named constant values of a nameset type. A declaration for a nameset-indexed object is like a variable declaration but there is a selector period and a nameset type name after the initial type name. The members are treated as variables in a simple object. For example,

```
nameset TrainingType coder, site_design, graphics, fonts, writer, editor
bool.TrainingType Skill
Skill.writer = TRUE
Skill.graphics = FALSE
```

This provides an easy way to identify bits, for example, as a 'mask' or 'flags' item.

The maximum value for a nameset constant name used in a nameset-indexed object is 63. In other words, values from 0 to 63.

A nameset-indexed object of type **bool** is packed into bits, for efficient usage. A nameset-indexed object of type bool can be type cast to **uint.64** or smaller **uint**. The value matching nameset constant number **n** is 2^**n**.

A pair of nameset-indexed objects of type **bool** can be used with the bitwise logical operators 'and' **&**, 'inclusive or' **|**, and 'exclusive or' **~**.

Example, using **TrainingType** nameset and variable **skill** from above:

```
# set a test mask -
bool.TrainingType TestMask
TestMask.writer = true
TestMask.fonts = true
TestMask.editor = true
# check skill - -
if (skill & TestMask).uint.isNonZero then @ "A writing skill is confirmed."
```

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

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 other name usages.

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. Values can overlap.

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 right-justified in a uint.64 double-word, with one bit unused.

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

Table: Standard Code Names

Code	Size	Values
ASCII	8	7 bit standard ASCII character values in 8-bit bytes
HEX	4	0:'0123456789' 10:'ABCDEF' 10:'abcdef'
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, same as **Windows-1252**
UTF8	8	UTF-8 characters (default)
UTF16	16	UTF-16 characters
UTF32	32	UTF-32 characters

These are defined in library **std/codes.ss**.

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

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. NUL (0x00) bytes can be anywhere.

Strings are not fixed length. There is no separate "character" type.

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

When a string constant is enclosed in quotation marks (but not apostrophes), object names with simple variable members and array elements with a simple index variable or an integer constant can be embedded by enclosing them in curly braces if they can be evaluated to a string. If a left curly brace is intended to be a character, precede it with a reverse slash.

Example:

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

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

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

Example:

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

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

String values are concatenated with a pair of vertical bars (||). String constants separated by a space or a continuation mark and end of line are also concatenated.

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

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

A string can be indexed like an array (first index is 0) to retrieve a single character (byte):

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

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

Table: Escaped Characters in a String

Escape Chars:	Meaning:
\n	End of line
\r	Carriage return
\f	Line feed character
\t	Tab character
\\	Reverse slash
\{	Left curly brace
\xdd or \Xdd	Hexadecimal value dd, 2 hex digits
\udddd	Unicode character: 4 hex digits
\Uddddddddd	Unicode character: 8 hex digits

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

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

A reverse slash before any other character is retained.

Regular Expression Pattern Strings

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

Curly braces in a pattern string do not allow insertions. The final slash has no following option letters. Options are specified with a mode.

Code Conversion in Strings

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

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

A type cast can also convert a code:

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

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

String Expressions

Concatenation of strings is performed with the operator ||. String constants separated by a space or a continuation mark and end of line are also concatenated.

Concatenation example:

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

A string acts like an array of characters when a subscript is used. The character referenced is extracted as a substring of one character. Thus, *stringvar*[*n*] identifies the single character number **n** (counting from 0) of *string*. This substring cannot be assigned a value because strings are immutable; it cannot be on the left of an assignment.

The characters of a string are numbered from 0, like arrays. The index counts bytes, not Unicode "code points".

Assignment copies a value.

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

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

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

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

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

Table: Built-in String Functions

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,s2)	Returns substring after **s1** and before **s2**, or **null** if none.
capsCase	String, the first letter upper case, the rest lower, in each word
nnn.Char	A character in LATIN1 code, from byte **nnn**
Clear	Reset string variable/array element to a zero length
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 invalid **m**
From(uint n)	String, starting with byte **n**, to the end, else **null**
getByteSize	Returns bits per byte/character
getCodeName	Name (a string) of the code
Last(uint n)	String returned, length **n**, from the end, else **null**
lowerCase	String returned with all letters (with accented) lower case
patn.replace(s, s2)	String s returned with all **patn** substrings replaced with **s**2
Reverse	String with characters reversed, preserving Unicode
Slice(uint m, 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, n)	Returns substring from character number **m** for **n** chars
titleCase	String, like capsCase except for words in array **_NonTitleWords**
trim	String returned with leading and trailing spaces removed
trimLeft	String returned with leading spaces removed
trimRight	String returned with trailing spaces removed
upperCase	String returned with all letters (with accents) upper case
upTo(uint m)	Substring from start through character **m**, **null** if none or bad **m**
str[n].Value	Value of the **n**-th byte of str, as a **byte**

The built-in string functions 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 (LF), \r (CR) and \t (TAB) codes

The case 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".upperCase 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

Table: Mathematical Functions

Func(y):	Base(x) Type:	Result Type:	Returns:
abs	float int	(base)	Absolute value
arccos	float	(base)	Arc Cosine
arcsin	float	(base)	Arc Sine
arctan	float	(base)	Arc Tangent
arctan(float **y**)	float	(base)	Arc Tangent of **y**/**x, y** is float
binary	(any uint or int)	string	Bits in the integer as a string
bitSize	(any uint or int)	uint	Returns 1, 8, 16, 32 or 64
ceil	float	(base)	Whole number > or =, away from 0
cos	float	(base)	Cosine
cosh	float	(base)	Hyperbolic Cosine
exp	float	(base)	Exponential, **e** to the power
floor	float	(base)	Whole number, truncated toward 0
hex	(any uint or int)	string	To a hex string, no leading 0x
isInfinity	float	bool	Test for "Infinity" + or -
isNaN	float	bool	Test for "Not a Number"
isNeg	int float	bool	Returns **true** if < 0 else **false**
isNonZero	int float	bool	Returns **true** if <> 0 else **false**
isPos	int float	bool	Returns **true** if > 0 else **false**
isZero	int float	bool	Returns **true** if =? 0 else **false**
log	float	(base)	Logarithm, base ***e***
log10	float	(base)	Logarithm, base **10**
maxValue(**y**)	float int	(base)	Maximum of base(**x**) and **y**
minValue(**y**)	float int	(base)	Minimum of base(**x**) and **y**
round(uint **y**)	float	(base)	Round **y** digits away from zero
roundEven(uint **y**)	float	(base)	Banker's rounding, even lowest digit
sin	float	(base)	Sine
sinh	float	(base)	Hyperbolic Sine
sqrt	float	(base)	Square root
tan	float	(base)	Tangent
tanh	float	(base)	Hyperbolic Tangent
!	byte	ulong	Factorial

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

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

In addition, the identifier **_PI** is defined as a float named constant with the value pi (π) to full float value in the automatically included math library. This identifier can be redefined; it is not reserved. Also defined are **_E**, as constant ***e*** and other values.

Typeset Specification

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

```
typeset arithType: int uint float
```

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

The typeset name arithType above represents a type which can be **int**, **uint** or **float.**

The specified types allowed also can be known typeset names or one of these specifications with special meanings:

object	any defined object type
*****	any basic type

A type preceded by the keyword **not** or **NOT** means it is not allowed. For example, the following *typeset_spec* allows a string or integer, but not **float**:

```
typeset key_type: string arithType NOT float
```

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

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

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

The standard library file **std/typesets.ss** defines several usable typeset names.

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

Procedures: Functions, Methods and Modes

A *function* returns a value, and thus can be used in an expression. It is invoked by naming an object on which it is defined, followed by a selector period, then the function name, then arguments, if any, in parentheses. The object is an implicit argument to the function. A function may have no arguments and no parentheses.

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

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

Functions, methods and modes (these forms are called *procedures* in SS) are defined in an object type definition, except when they are based on **null** or a basic type or known nameset type they can be defined outside an object.

Passing Arguments to Procedures

A procedure may be invoked upon an object. The object is then implicitly passed as a reference argument internally referred to as **$**.

The procedure name is applied to the object by placing the procedure name after a selector period on the right of the object, no spaces around the period.

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

A method never uses parentheses on its argument list, even when arguments are passed.

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

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, arrays, lists and functions.
- A new object type definition.
- A **nameset** definition.
- A **nameset-indexed** object declaration.
- A named constant declaration.
- A **typeset** declaration.
- An **interface** declaration.
- A procedure (**method**, **funct**, **mode**) definition.
- An indexer declaration.
- A constructor declaration.
- An operator declaration.
- A type cast declaration.
- A **code** declaration.
- An **pragma** declaration.

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

Declaring Variables and Arrays

Places where variables, arrays, properties and lists can be declared:

- In a block, also a switch case block.
- In an object definition.
- In a procedure definition.
- Variables for the index and value are implicitly declared, with limited scope, in a for construct. The index is the index type of the array. The value is the same type as the array.
- Variables are declared with type information before them. Types are bool, uint, int, float, string, any nameset type and any known object type name. Examples:

```
int number
string name
```

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

```
int mnx = 23
string ap[] = ['start','stop']      # indexes are 0, 1
uint.64 salary[string] = ['CEO':250_000, 'analyst':45_000]
```

- 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.
- As long as the type is the same and the access is the same, variables, arrays, lists and properties (public only) can be defined in one declaration:

```
short NumPartTypes, Parts[], PartTypes list, PartsRemaining return Parts.Count
```

Named Constants

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

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

The *access* keyword is used only in an object type definition. Access can be private in a block.

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 functions or functions. It must be convertible to the declared type.

Examples:

```
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)
```

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

Statements

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

Statements are only allowed in a procedure, including a method, mode, function, generator function, operator definition, indexer, type cast definition or function definition.

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, the block construct, subordinate statements **case**, **continue**, **else, final**, **when**, **while**, **until** and the control statements **return**, **if**, **break**, **repeat** and **signal**.

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

- The *assignment* statement uses the assignment operators **=** and *op***=**.
- The *incrementation* statement is actually a simple expression using the postfix incrementation (**++**) or decrementation (**--**) operator. The side effect change makes this also a statement. It is considered an assignment when it stands alone.
- An array assignment.
- A method call is a statement.
- A mode invocation can be a statement if it stands alone, not in an assignment or expression.
- A construct is considered to be an executable statement.

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

If Statement

An **if** statement has the form:

```
if bool_expression statement
```

where *bool_expression* is an expression that evaluates to **true** or **false**, and *statement* is a single executable statement, not **if** and not a construct. The statement can be an assignment or incrementation or a method call or mode call, which must be preceded by **then**, or it is one of the statements **break**, **repeat**, **signal** or **return**. It cannot be a block, **if**, **while**, **until** or **continue**.

If Construct

The *if_construct* has the following form:

```
{ if bool_expression
      statements
[else if bool_expression
      statements]...
[else
      statements]
[when
      statements]
}
```

A *bool_expression* is an expression that evaluates to **true** or **false**. There is no statement or colon after the *bool_expression*.

Keywords **else** and **when** are not statements, but begin sections of the construct.

Additional conditions (**else if**) are allowed after the first one, and before optional **else**. If the first **if** condition fails, it tries the next, etc., then the **else** section is executed if all fail. These have the word **else** before **if**. Only one of the conditions is executed, the first to be true.

Note that the similar-looking **if** statement is considered a single statement, not part of a construct.

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

Switch Construct

The *switch construct* provides a selection of alternative actions, somewhat like a conditional block.

It has the following form:

```
{ switch expression
case constant | constant_range [,...]
      statements
[case constant | constant_range [,...]
      statements]...
[else
      statements]
[when
      statements]
}
```

The expression on the **switch** must be an integer or string or nameset valued or defined object expression. It may not be an array, nor if it is a defined object, may any part of it be an array. The constants on the **case** lines must all be the same type as the switch value. The expression is evaluated exactly once in a switch statement.

Execution proceeds to the **case** or **else** which matches the value of the expression. The following statements (the *case_block*) are executed. Execution goes to the ending curly brace if it falls through the statements into a **case** or **else**, rather than fall through into the following case block. The **continue** statement overrides this, allowing fall-through. This is not like the C or Java or similar languages.

The **break** or **continue** statement can be used to override the flow of execution. A **break** sends control to the end, and **continue** before **case** or **else** overrides the implicit jump to the ending curly brace. The **continue** is allowed only as the last statement before **case** or **else**.

No statements are allowed between **switch** and the first **case** or **else**. There must be as least one case block in the construct, not just **else**. Statements which are not allowed in a switch block or case block except inside a contained looping block: **until** and **while**.

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. The range feature is not allowed for nameset or defined object constants.

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

The **else** block (which is like a case block) indicates selection of any other unspecified value. It must follow all **case** sections.

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

Example of a switch:

```
int mv = 8, xlv
{switch mv
case 1 : 5
   xlv = 0
case 6:25, 50:59
   xlv = 1
else
   xlv = -1
}
```

Block Construct

The *block_construct* is a block which does not loop. It does not contain a **final** block or **begin** block, and there are no looping control flow statements (**repeat** or **while** or **until)** except **break**, which may be conditional. It has the form:

```
{
        statements
[when
        statements]
}
```

A block with no loop:

```
{ int jxx = 1
  statements
}
```

A non-looping block may have declarations which are local in scope. The first declaration or statement (not a construct) may be on the same line as the initial curly brace.

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

Looping Block Construct

A block is a looping (repeating) construct if there is a **repeat** or **while** or **until** or **break** statement in the construct. It may also have a **begin** block and a **final** block.

```
{
[   initialization statements
begin]
    statements in the block
[final
    statements]
[when
    statements]
}
```

The statements between the left curly brace and the **begin** line are a *begin_block*, or initialization block. These statements are executed once only. Any variable defined in the initialization block have scope over the entire construct.

The statements after **final** are executed every time through the loop. A **repeat** sends control to the final block. This is where an incrementation is placed.

Control flow goes back to **begin** from the end of the final block (or right curly brace). If there is no **begin**, it returns to the left curly brace.

Example of a loop:

```
{  int k = 1024
begin
   until k =? 0
   # statements here
final
   k = k/2
}
```

This loop assigns a value to **k** once before the **begin** declaration, then repeats the statements in the block, dividing **k** by 2 at the end, and going back to **begin**. Since type **int** for **k** is specified, the variable **k** has local scope in the construct. The loop terminates when **k** is zero.

A simple loop example:

```
{  while test_expression
   statements
}
```

There is no **final**, **begin**, **until**, **repeat** or **break** in this looping block.

A looping block cannot be a procedure or if block.

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

Initialization (Begin) Block

The first lines after the left curly brace of a block construct can be in a **begin** block, which provides an initial value. A **begin** block consists of declarations and statements ending with line **begin**. The **begin** block alone does not make the block loop. This initialization is not repeated in a loop because it is before a **begin** line. A **begin** is not valid in any other block type.

Only one **begin** is allowed in a block.

If *var* is declared with a *type* in a **begin** block the variable has that type and has scope local to the construct and the initialization is done once at that point. The value is not retained outside the block.

If *var* is not typed and it is assigned a value, it must be a valid existing typed variable, and the value is retained after the end of the construct for its normal scope.

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

While and Until Statements

While or **until** may be placed in a looping block or **for** block anywhere before the **final** or **when** block and after a **begin** declaration.

While

The **while** statement has the form: **while** *test_expression*

The *test_expression*, an expression of type **bool**, is evaluated and control does a **break** to the **final** block or the ending curly brace if the expression is **false**. The presence of a **while** makes the block loop. The **while** statement is treated as:

if not (*test_expression***) break**

Until

The **until** statement has the form: **until** *test_expression*

The *test_expression*, an expression of type **bool**, is evaluated and control does a **break** to the **final** block or the ending curly brace if the expression is **true**. The presence of an **until** makes the block loop. The **until** statement is treated as:

if *test_expression* **break**

These statements are valid only in a "looping" block, not an **if** block, a **switch** block, a **for** construct, a procedure block or **when** block.

If the **while** or **until** is the first statement or declaration after the curly brace, it may be placed on the same line.

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

For Construct

The **for** construct loops through an array or string or an object with certain capabilities, handling each value. There are several forms of the **for** construct. The first line differentiates which form is used.

```
[1]   { for val[[k]] of array_expression
            block
      [final
            block]
      [when
            block]
      }
```

The expression is evaluated once. Then each element of the array is extracted, assigning the (local scope if not already typed) variable *val* the value, repeating the *block*. The type of *val* defaults to the same as the array type. Only the array elements which exist (the index exists) are used. Values are presented in index order, with or without the index **k**.

Example of [1]:

```
float cost[string] = float['notebook':2.55, 'pen':0.77, 'paper':1.98]
{ for cost[item] of prices
  @ "item={item}, cost={cost}"
}
```

A string can be used also, similar to an array. Variable *val* will be assigned each character, left to right. Unicode characters are identified.

```
[2]   { for val [[k]] of string_expression
            block
      [final
            block]
      [when
            block]
      }
```

The expression is evaluated once. Variable *val* will be assigned each character, left to right. Unicode characters are identified. The local scope variable **k** is assigned the byte index value of the character and implicitly local (or previously declared) variable **val** gets the value, type **string** and same code as *string*, repeating the *block*. The type of optional variable **k** is **uint**, and **k** will begin with 0. Variable **k** has local scope to the block or it may be pre-declared. The string cannot be altered in the block.

```
[3]   { for val of _Traversable_object
            block
      [final
            block]
      [when
            block]
      }
```

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

A nameset-indexed object acts somewhat like an array in a **for** block:

```
[4]   { for val [[k]] of nameset-indexed-object
            block
      [final
            block]
      [when
            block]
      }
```

This returns each value *val* and (optionally) the nameset index name as the implicitly declared string variable **k**. The value of string variable **k** can be found by appending a period then the nameset type name.

A list acts somewhat like an array or string in a **for** block:

```
[5]   { for val of list
            block
      [final
            block]
      [when
            block]
      }
```

This returns each value *val*, of the list.

Finally, a generator function can be used.

```
[6]   { for val of object.generator_function
            block
      [final
            block]
      [when
            block]
      }
```

This repeatedly calls the *generator_function* returning a value until it returns **null**, which causes a break.

Final Block

A *final_block* can be used in any looping construct or **for** construct.

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

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

The *final_block* is executed on every repetition of the loop**.**

An empty *final_block* can be omitted.

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

A **break** statement anywhere in the construct exits the loop.

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

Repeat Statement

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

Break Statement

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

Example:

```
int i   # scope extends after the block - -
{ i = 0
begin
  until i >= 10 OR price[i] < 0
  total += price[i]
final
  i++
}
if i<10 then @ "Error for price number {i}"
```

Continue Statement

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

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

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

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

Return Statement

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

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

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

In a method, the return statement has no parameter. A mode can omit the parameter if the base type is returned, or supply a result 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 returned value of the function.

In a function, the last statement in the function must be a return with a value.

A generator function which has control falling into the end of the block implicitly returns **null**.

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

Signal Statement

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

signal [*status*]

where *status* is a value of the type **_Status**, which is a predefined object type, in **std/cond.ss**. The status must be named on the signal statement except in a **when** block; in a **when** block, a signal with no status named passes the current status to the next outer level **when** block.

Terminating a Program

To terminate execution of the program, the **signal** statement is used, with a status which will be handled by the **@Main** constructor in a **when** block, or which will go to the system default handler. A defined status can pass a desired message text.

The following is a suitable signal for an exit:

signal _STAT_End

A shared definition of nameset _StatusLevel and object type _Status with status constants is in **std/status.ss**,

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

When Block

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

A *when_block* can appear at the end of any block, for block, a procedure block (including constructor, type cast, operator definitions), or a switch construct, or an if construct. It is not part of an object definition. A *when_block* must not be empty.

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

Upon entering the *when_block*, the static variable **_Status.Current** has been set with the value of the signaled status.

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

signal _Status**.**Current

or doing nothing, or simply:

break

If the variable _Status**.**Current is set to **null**, control passes to the program termination.

If no **when** block handles it, a system-supplied action is performed for the status.

A *when_block* can contain a switch block on the value **_Status.Current** or other actions.

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

Control Flow Limitation

No executable statement may appear immediately following an unconditional control statement which breaks the normal flow of control.

In other words, a statement is not permitted after **return**, **break**, **continue**, **repeat** or **signal**.

This rule does not apply to the return statement in a generator function, nor to the above statements on an **if** statement.

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

Blocks

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

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

Blocks are:

- The *for construct* or *block construct* includes the entire construct and the block executed. It also includes the *final_block* and *when_block*.
- The *case_block* is all of the statements and declarations from case or else in a switch until case or else 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 else blocks. Declarations have scope in the switch block where they appear, including the when block if one is used.
- The *conditional_block* is all the statements in the braces.
- The *procedure_block* is all statements and declarations in a procedure definition. This begins at the start of the prototype portion and ends at the procedure end. This applies also for operator and type cast definition and for a constructor. It also applies to variants.
- The *final_block* is the statements after **final,** in the following block. It is not a true block because it is part of the same scope as the construct.
- The *when_block* is the statements from **when** through the following block to **end**. Its scope extends the preceding block. A *when_block* follows a **final** block in a block or each construct. It is permitted at the end of any construct and a procedure (**funct**, **method**, **mode**) block.

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

Defining Objects

An *object* in SS is a named entity which has values, procedures and other functions within it. Objects are used to encapsulate data and functions, 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 functions. Multiple inheritances are disallowed.

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

```
{ [final][implicit] object object_type [inherits object_type [uses variant_list]][enables
interface_list]
    [typeset declarations]
    [nameset declarations]
    [variable and array and list declarations]
    [indexer declaration]
    [procedure definitions]
    [operator definitions]
    [type cast definitions]
    [abstract procedure/indexer/function/operator/type cast/property definitions]
    [constructor definitions]
    [pragma declarations]
    [code declarations]
}
```

Object definitions cannot be directly nested or internal.

An object type with the option **final** cannot be inherited.

The option **implicit** defines an object that cannot be inherited explicitly, but is used by the language for predefined purposes. These object type names are restricted and they begin with an underscore.

The **enables** and **inherits** phrases may appear in any order. The **uses** part is considered part of the **inherits** phrase.

Variants listed after **uses** must exist. There must not be more than one named which are based on the same procedure.

A *variant_list* is a set of variant names separated by spaces.

An *interface_list* is a set of interface names separated by spaces.

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

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

Visibility of Names - Access

Access is one of the keywords **private**, **protected**, **public** or **static**. if unspecified, it is **public**. It controls the places where a name of a variable, array, procedure, function or an object are valid, or visible.

Static

A variable, array or procedure 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. A static variable or array is public. For example, static value **val** in object type **ABC** is referenced as:

```
ABC.val
```

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

Static procedures do not have an instance. They can only use static values.

Final

A value or function (or procedure) with the option **final** cannot be overridden.

Private

The keyword **private** means visible in the current object only. It means the name cannot be inherited.

Protected

A protected name is visible in the object and in inheriting objects, but not outside. It is not public.

Public

When the access is not specified, the item is public and is visible across the object and any inheriting objects and the member is visible as a member of the object.

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

Variable, Array and List Declarations

Variables and arrays are defined in an object in the form

[*access*] *type_name* name [*array_spec* | **list**] [**=** initial_value]

Variables and arrays and lists in an object are public if no access is specified.

Definition examples:

```
protected float rate = 0.035
string students[string]
uint sizes list = (4, 6, 8, 10, 12, 14)
# if they are the same type and access, then
# variables and arrays can be on the same line - - -
byte abc, xyz[]
```

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

Indexers

An *indexer* is similar to a function. It provides a way to define the access of array elements of any array of the object type in which the indexer is defined. An indexer is an accessor.

The form of an indexer is:

```
[index_type [index_name]] [expression]
```

The *index_type* and *index_name* are used like a subscript. Index type is a basic type suitable for an array index. The *index_name* and expression are omitted in an abstract indexer. An abstract indexer can be in an interface, with similar rules to a function.

The expression must be present except in an abstract indexer or an interface prototype. It is a reference to the data portion of the indexed member of the array. It finds the element, and if it does not exist, creates a member and inserts it, with default data, and returns the reference.

There can be several indexers in an object type, one for each permitted index type.

An indexer cannot be defined for a **list**.

A typical use of an indexer is to define the array as members of a tree. It may also define rules for indexing an array using buffering or hashing.

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

Object Values and Constants

Object values are constructor calls with arguments and values.

If the values assigned are constant, the result is an object constant.

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

Example:

```
{ object Animal
  string kind
  string says
}

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

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

Shared Definitions

Shared definitions are defined only outside any object type definition. In addition to objects, they can be:

```
[typeset declarations]
[named constants]
[nameset type definitions]
[code declarations]
[interface definitions]
[methods based on null]
[functions and methods on a basic type, defined object type, a nameset type
 or a typeset]
```

The names defined obey normal scope rules. They are global names, as if the shared definition is inherited by every object. There are no shared variables or arrays or functions.

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

```
const float _PI 3.14159265358979323846264338327795 # ...
const float _E 2.71828182845904523536028747135266 # ...
const float _LN_10 2.30258509299404568401799145468 # ...
const float _ONE_DIV_LN_10 0.43429448190325182765 # ...
```

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

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

Modes, indexers for an object and constructors cannot be defined outside an object definition.

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

Procedures defined outside an object or its inheritors but based on the object can only use public names in the object.

Inheritance

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

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

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

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

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

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

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

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

Enables Specification

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

The *enables_spec* has the format:

```
enables interface_name ...
```

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

Instantiation of an Object

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

Default Value at Instantiation

An integer, float variable has a default value of 0.

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

A bool variable defaults to **false**.

Object Member References

The object name is followed by a period, then by the function name or the procedure name.

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

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

Enhanced Type Objects

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

The enhanced object is restricted in several ways:

- It can contain only one non-static data member, of the same type as is inherited.
- It can contain static members, also constants.
- It can contain functions and procedures.

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

The enhanced object type is a way to specialize values, One enhanced type cannot be mixed with another even though they both inherit the same type. For example, a Meters type and an Inches type can inherit float, but they cannot be mixed by mistake. Functions 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 type cast a metric Length to inches by using **Length.float.inches**. The correct way is to provide functions or procedures which convert directly.

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

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

Properties

A property looks like a variable in usage. It is a public value, defined only in an object. It is based on the object. In some ways it resembles a function with no parameters.

A property can be read-only (return only, like a function), write-only, or both read and write, like a variable.

A property abstract is allowed in an interface.

A property definition is:

type [*****]*name* [**return** *expression*] [**set** *assignment_or_procedure_call*]

where:

- *name* is the property name.
- *type* is the type returned or set.
- *expression* is the value returned or assigned. If these are omitted, the property is an *abstract property.* Abstract properties are allowed in interfaces or an object definition, which then becomes an abstract object.
- The set phrase must use the property name as a variable.
- The set phrase and the return phrase can be in either order.
- Property definitions can be on the same line with variable, array, and list definitions, with a single shared type name, separated by commas.
- If the asterisk appears before the property name, it alters the definition of the name, but it is allowed only when it begins with an underscore. In that case, the name, without the underscore, is public, as a return-only property, and it remains available as a "protected" property with the underscore, also with the set (write) usage, inside the object.

Example of Property Usage

Allow conversion between inches and centimeters:

```
{ object Dimens
  private float wid, len # inside the object, values are inches, outside is cm
  public string item_name
  # 3 properties - - -
  float width return wid*2.54 set wid = width/2.54
  float length return len*2.54 set len = length/2.54
  float area return wid*len/(2.54*2.54) # returns square cm
}

Dimens door
door.item_name = 'Rear entry door'
door.width = 120 # centimeters
door.length = 300 # centimeters
@ "{door.item_name} is {door.area} square cm in area."
```

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

Defining Procedures

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

A procedure (not a mode) can be defined on an array of any type by specifying an object type followed by a pair of square brackets. The array index type can be specified to apply to a specific type of index. The array type must define indexing.

A procedure (not a mode) can be defined on a list.

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

The characters -> indicates the returned type in a procedure definition.

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 function 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 type can be **self** or a basic type name or **null**.

The base name can also be the name of another object type.

Procedure Overloading and Overriding

Within an object definition, procedures can have the same name, but differ in their parameter specifications. One function **Fun** may have an **int** argument while another **Fun** has a float argument. This is called overloading.

An object may inherit another object, and both define the same procedure. The inheriting object is said to override the inherited procedure. If a procedure is abstract, there must be an overriding definition in an inheriting object.

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

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 when an object is defined by placing its name in a list after the keyword **uses** following the object type name.

A procedure with a variant cannot be abstract.

A variant of a generator function is also a generator function.

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

The variant assumes the same base type. The arguments, if present, must match the same order, type and name as the base procedure, but default values can be different. If not specified, the arguments and values are assumed to be the same. The result type must match if the variant is a function. A result as an array cannot have different indexing.

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

A variant name procedure is not referenced as a procedure by name anywhere in an object definition, nor invoked outside the object. Choosing a variant merely substitutes its body and arguments for the overridden procedure.

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

A variant name can apply to multiple procedures in an object.

Asterisk before a Procedure Name

This feature is allowed only in an implicit object type.

If an asterisk appears before the procedure name, it alters the definition of the name, but only when it begins with an underscore and is protected or private. In that case, the name, without the underscore, is public. It remains available as a protected procedure with the underscore, inside the object.

This allows a procedure to be available for the user but protected for system purposes.

This feature is used for defining array indexing defaults, for example.

Method

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

```
{[proc_option][access][base_type] method [*]name [args_spec]
      block
 [variant name [args_spec]
      block] ...
 [when
      block]
 }
```

Function

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

```
{[proc_option][access][base_type] funct [*]name [(args_spec)] -> type [return_val]
      return_block
[variant name [(args_spec)] -> type [return_val]
      block] ...
[when
      block]
}
```

The *proc_option* keyword **repeat** implies it is a *generator_function*.

The *args_spec* (including parentheses) is omitted if there are no arguments.

The *type* is the function (returned) type. It is required.

The *return_val* is an optional returned value expression. If the returned value is shown, no function return block is used, and if variant and when are not needed, the surrounding curly braces can be omitted.

The simplest form of a function can be defined in a single line:

```
[proc_option][access][base_type] funct name -> type return_val
```

Mode

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

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

```
{[proc_option][access][base_type] mode [*]name [(args_spec)] [-> return_type]
      block
 [variant name [(args_spec)] [-> return_type]
      block] ...
 [when
      block]
 }
```

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

Procedure Option

The *proc_option* on the start of a procedure definition may be any of the keywords **repeat**, **final** or **static**. These precede the *access* keyword, if it is supplied.

Specifying **repeat** is allowed only on a function, implying a generator function.

Procedure Prototypes

If the *body* and the surrounding curly braces are omitted on a procedure, or if a function definition has missing actions for **return** or **set** in an object definition, it is an abstract prototype. 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.

An object type is considered abstract if it contains an abstract item. It must be inherited and the abstract item must be defined.

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

Final Procedures

A procedure with the option **final** cannot be overridden.

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

The Return Specification

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

```
result_type [return_value]
```

The *return_value* is a value expression to return. It may use constants and other values known in the object or arguments of the function. Variables in the instance are accessible.

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

The *result_type* or *arg_type* or *base_type* can be a basic type such as **int**, **uint**, **float**, **bool**, **string** or it can be an object type name, and it can be an array specification (with type and index type), a **nameset** type name, or a **typeset** name.

When *base_type* is an object type, the procedure has access to protected or public internal data and properties and procedures inside the definition and data shared from an inherited object type. Private data is hidden unless the procedure definition is inside the object.

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 **null** as an indicator of failure, meaning it failed to create the object, or the end of the generator results.

Basic_Object_Type Procedures

A *basic_object_type* procedure is not based on a defined object type or nameset type. The base_type can be a basic type (**int**, **float**, etc.). Methods and functions can be defined, modes cannot.

A method can be defined on **null**, but a function or mode cannot be on **null**. If a method is defined on *base_type* **null**, the invocation has no base.

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

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

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 basic-object method.

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

Procedures in an object are defined with any appropriate access.

Access

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

Function items are like variables in appearance, but may in fact be implemented as a procedure, or they may be implemented as a public variable if the compiler finds this is safe. They are possibly limited to read-only or write-only access.

The Base Type

The *base_type* on a procedure definition is omitted or is **self** when it is the containing object type.

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

Arguments Specification

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

The *args_spec* is omitted fwhen there are no formal arguments, or it is:

```
arg_type [ref] arg_name [= constant_expression]
```

for each argument. Arguments are separated by commas. The argument type a*rg_type* can be omitted if it is the same as the previous *arg_type*.

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

All value variables (types int, uint, float, bool) are passed by copy, unless the keyword **ref** is used. An array element is passed by copy unless the argument is marked **ref**. The array is not changed when a copy is passed.

A variable passed by copy delivers a temporary copy to the procedure. If the procedure alters the copy, the original is preserved.

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

An expression always passes a copy. An item enclosed in parentheses is a simple expression. An array value or constant passes a copy. A list value does the same.

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

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

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

The Procedure Body

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

In an interface declaration, the body is omitted, leaving only the prototype.

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

Table: $ and self

$	The current instantiated object, used as a variable
self	The current object's type

The name **$** can be an Lvalue, but the actual passed base is not altered.

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

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:

```
{ mode begin [(args_spec, ...)]
        block
[when
        block]
}
```

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

The argument names may be the same as public names, or they must appear in the constructor's block as references in an expression.

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.

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

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 procedure block (before **when**) cannot be empty.

Default Constructor

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

If there is no constructor, a default one is provided, with arguments which are the names of the public variables and arrays and properties.

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

Generator Functions

A generator function is defined with the keyword **repeat** as a *proc_option*.

It can be used only in a **for** construct or the right side of an array assignment..

A generator function returns ("yields") a value using the **return** statement. Each time it yields 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 **return null** or the function end, which then returns **null**.

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

Range

A generator function **Range** is supplied. It returns a sequence of **long** values. It is based on **null**, with these three **long** arguments:

start	default is 0.
by	default is 1; if negative, **stop** must be less than **start**.
thru	required; last value. Range stops beyond this value.

If these are invalid, unable to produce a value, Range signals **_STAT_InvalidRange**.

Examples:

```
int ABC[] # longs convert to int
ABC = Range(thru=9) # sets array elements to 0 through 9
ABC = Range(start=1,thru=10) # sets values 1 through 10
ABC = Range(start=2, by=3, thru=13) # values 2,5,8,11 but NOT 14
ABC = Range(start=5, thru=-5, by=-1) # values 5,4,..,-4,-5
```

Invocation of a Procedure

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

Invocation of a method will not use parentheses.`

A mode or function invocation does not use a pair of parentheses when it has no arguments. A sequence of applied procedures (modes or functions) is called a *stack*. The right-most can be a method.

If there is no base_type (it is defined on **null**), the base and the period are omitted.

When a function or mode returns a reference to an object, that object may have other procedures or 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 **$**, which is passed with an implicit **ref**. When the base is a constant, a basic type value or an 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.

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

Built-in Object Procedures

These procedures or properties can be applied to any object, variable, list or array, treating them as objects.

Table: Built-in Object Procedures

Procedure	Kind	Result
Clear	method	the named object is destroyed
Copy	mode	a copy of the item
getIndexType	property	String, the type name of the array's index
getName	property	String, nameset constant name for value in variable
getType	property	String, name of the type of the object
getValue	property	**uint** value of the nameset constant
getValuesArray	property	returns string-indexed array of values in the nameset type used as base
isNotNull	property	**true** if the ref to the object is NOT **null**
isNull	property	**true** if the ref to the object is **null**
isA(string **t**)	property	**true** if the object type or its inherited type matches **t**
testNull(val)	function	returns **val** if base is null, else returns the base; **val** must be the same type as the base

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

The functions **getName**, **getType**, **getIndexType**, **getValue** and **getValuesArray** are reflection functions. Reflection in SS is the ability to retrieve attributes of an object. Reflection is limited in SS to a these functions.

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

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

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

Name References in an Object

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

Example:

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

{ object DEF inherits ABC
  string x = 'World'
  { mode begin # constructor
    @ "{ABC.x}, {x}!"
  }
}

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

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

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

Defining Operators

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

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

```
{[base_type] funct op [(type b)] -> return_type [return_value]
        block
 [when
        block]
}
```

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

An operator definition is public.

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

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

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

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

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

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

Assignment cannot be defined.

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

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

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

A simple definition can be in one line:

```
Obj funct +(float y) -> Obj (Obj.val + y)
```

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

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.return_type**, where **abc** is the new type *base_type*:

```
{ base_type mode -> return_type
      block
[when
      block]
}
```

Or, the simple form:

```
base_type mode -> return_type returned_value
```

If the returned value is shown on the first line no block or **when** is used and the left brace is omitted.

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

A type cast definition is public.

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

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

An array or a list can be type cast if the members can be the new type.

Example: a simple definition, in one line:

```
Obj mode -> string "${Obj.val.string}"
```

Interfaces

An *interface* is a declaration which defines the prototypes of a list of object functions, methods, modes, type casts, indexers, properties and default named constants. The *prototype* of a procedure is the initial declaration without the body part or the implementation part.

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

An interface definition may appear in or outside an object.

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

An interface declaration has this format:

```
{ interface interface_name [enables interface_list]
   [typeset]
   [procedure prototypes]
   [type cast prototypes]
   [indexer prototypes]
   [default named constants]
  [indexer prototypes]
}
```

An object type can enable multiple interfaces.

Unlike an object declaration, there are no variables, arrays or lists 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 override any part of the enabled interface.

An object that enables an interface must complete the implementation of all items in the interface and those inherited by the interface. Instead of completing the implementation, an object can leave a procedure as abstract.

Example of an interface:

```
      # a double linked list
      { interface _TwoWayTraversable enables _Traversable
         # inherits Next (etc.) from _Traversable
# ???????????????
         funct Prev # backward link
         bool hasPrev
      }
```

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.

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

For example, the following interface:

```
{ interface Smaller
   typeset num_type: int uint float
   num_type funct Reduce return num_type
}
```

can be enabled (implemented) by an object that uses integer or float:

```
{ object Obj enables Smaller
  { num_type funct Reduce -> num_type $ - 1
  }
}
```

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

Standard Interfaces

Special interface definitions are supplied in standard library **std/interfaces.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
  funct =?(_DataType d) -> bool
}
```

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

_Comparable Interface

The interface **_Comparable** is defined as:

```
{ interface _Comparable enables _Equality
   funct >(_DataType d) -> 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.

_Array Interface

This interface defines the functions of an array with indexing.

```
{ interface _Array
  [_KeyType]
}
```

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

_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
   _Traversable First
   _Traversable Current
   _Traversable Next
   hasNext -> bool
}
```

When there is no traversable **Next** member, the Next return value is **null**. Also, **hasNext** tests for **null**.

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

```
{ for item of obj
   block
}
```

internally expands into:

```
{ _Traversable item = obj.First
begin
   block
   while item.hasNext
final
   item .= Next
}
```

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. The '**final**' is needed in case the block requires it.

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

Prototypes

Prototypes define the usage of procedures, properties, operators, indexers and type casts. They are used in interfaces and in abstract object definitions.

Function Prototype

[**repeat**] *base_type* **funct** [*]*name* [**(***args_spec***)**] **->** *result_type*

Method Prototype

base_type **method** [*]*name* [**(***args_spec***)**]

Mode Prototype

base_type **mode** [*]*name* [**(***args_spec***)**] [**->** *result_type*]

Type Cast Prototype

base_type **mode ->** *result_type*

Operator Prototypes

base_type **funct** *operator* [**(***args_spec***)**] **->** *result_type*

Indexer Prototype

base_type **[***index_type***]**

Property Prototype

result_type [**set**] [**return**]

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

Pragma Declaration

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

Some features to be implemented:

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

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

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

Example of **pragma adapt** -

```
{ pragma adapt sinh # assumes function, config names its location
  funct sinh -> float # using external notation
}
```

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

Printing Strings

The standard library member **std/std.ss** defines some built-in procedures. One method which is defined is named **@**. 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 with end-of-line character(s) added to the end of the output string.

The method **@noEOL** prints the string without adding end-of-line.

Example:

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

Note the string insertion in the message.

These method names are not reserved words. They are defined as single argument methods based on **null**.

Hello, World! Program

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

```
{ object @Main
   { mode begin
     @ "Hello, World!"
   }
}
```

It is automatically invoked by its constructor:

```
@Main
```

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

Regular Expressions

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

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

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

Example:

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

Table: Built-In Functions for Regular Expressions

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

Optimization of Regular Expressions

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

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

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

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 may be yet another set of formats when they are used with databases.

Also, 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.

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

Complex Data Type

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

The following functions or 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
Conjugate	complex	Complex conjugate
cos	complex	Cosine
cosh	complex	Hyperbolic Cosine
exp	complex	Exponential, **e** to the power
Imaginary	float	Imaginary part
isNonZero	bool	Returns true if <> 0 else false
isZero	bool	Returns true if =? 0 else false
log	complex	Logarithm, base ***e***
log10	complex	Logarithm, base **10**
Real	float	Real part
Reciprocal	complex	Complex 1.0/x
sin	complex	Sine
sinh	complex	Hyperbolic Sine
sqrt	complex	Square root
tan	complex	Tangent
tanh	complex	Hyperbolic Tangent
Times_I	complex	Multiply by the imaginary ***i***

Property **isZero** is **true** if both real and imaginary parts are zero.

Property **isNonZero** is **true** if either real or imaginary is nonzero.

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

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

Example - Polymorphism

This example is patterned after a set of sample programs showing 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 enabling an interface SaysWhat:

```
{ interface SaysWhat
  string Name return
  method Speak
}
# the base object Animal - - -
{ object Animal enables SaysWhat
  string animal_name
  string Name return animal_name.capsCase set animal_name = Name
  { method Speak
     @ "{Name} says \"I am an animal.\""
  }
}
# animals Cat and Dog
{ object Cat inherits Animal # overriding method Speak for Cat - - -
  { method Speak
     @ "{$.getType} {Name} says \"Meow!\""
  }
}
{ object Dog inherits Animal # overriding function Name for Dog
  string Name return "Canine: {animal_name.upper}"
}

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

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