The Sexy Scripting Language

By Mark Anthony Taylor

(Otherwise known as SexyScript, Sexy or Rococo-Sexy)

# Contents

# [The Basics](#_The_Basics)

# [S-Expressions](#_S-Expressions)

## [Atomic Expressions](#_Atomic_Expressions)

## [String Literals](#_String_Literals)

## [Compound Expressions](#_Compound_Expressions)

## [Root Expressions](#_Root_Expressions)

## [Comments](#_Comments)

## [Functions](#_Functions)

## [Inputs](#_Inputs)

## [Outputs](#_Outputs)

## [Primitive Types](#_Primitive_Types)

## [Struct Types](#_Struct_Types)

## [Namespaces](#_Namespaces)

## [Aliasing](#_Aliasing)

## [Using Directives](#_Using_directives)

## [Interfaces](#_Interfaces)

## [Classes](#_Classes)

## [Constructors](#_Constructors)

## [Destructors](#_Destructors)

## [Class Factories](#_Class_Factories)

## [Invocation](#_Invocation)

## [Invoking Functions](#_Invoking_functions.)

## [Invoking Local Variables](#_Invoking_local_variables)

## [Assignment to local variables from functions](#_Assignment_to_local)

## [Get-Accessors](#_Get_Accessors)

## [Invoking Interface Variables](#_Invoking_Interface_Variables)

## [Interface Assignment from a Function](#_Interface_Assignment_from)

## [Invoking Methods](#_Invoking_Methods)

## [Method Get-Accessors](#_Method_Get-Accessors)

## [Exceptions](#_Exceptions)

## [Throwing](#_Throwing)

## [Catching Exceptions](#_Catching_Exceptions)

## [Nested Exception Handlers](#_Nested_Exception_Handlers)

## [Primitive Algebra](#_Primitive_Algebra)

## [Implicit Parentheses](#_Implicit_Parentheses)

## [Structures and arithmetic operators](#_Structures_and_arithmetic)

## [Structures on the stack](#_Structures_on_the)

## [Delta Operators](#_Delta_Operators)

## [Control-Flow](#_Control-Flow)

## [If-Statements](#_If_Statements)

## [While-Statements](#_While_Statements)

## [Do-While-Statements](#_Do…while_statements)

## [For Statements](#_For_Statements)

## [continue](#_Keyword_continue)

## [break](#_Keyword_break)

## [return](#_Keyword_return)

## [goto / label](#_Keyword_goto_/)

## [yield](#_Keyword_yield)

## [finally](#_Keyword_Finally)

## [Boolean Valued Expressions](#_Boolean_Valued_Expressions)

## [Boolean Valued Compound Expressions](#_Boolean_Valued_Compound)

# The Basics

## S-Expressions

All sexy script files are human-readable plain-text and must conform to the Sexy S-Expression syntax. S-Expressions were invented before programming languages, back in the 1950s. The first language to use them was LISP. Today they have fallen out of fashion, but the advantages never went away and have been resurrected for our new language.

An S-Expression is a sequence of elements separated by blank space characters. Elements can be string literals, atomic expressions or compound expressions, with the later enclosed in parenthesis.

Sexy S-Expression syntax has a few rules on top for specifying such as string escapement and comments. Currently only ASCII characters are valid, though S-Parsers should generally not complain if characters are present from codes 128-255.

Blank space characters have ASCII values 1-32, and they demarcate elements. The null character with ASCII 0 generally means end-of-file, and it should not appear in a plain-text document.

## Atomic Expressions

The simplest element is an atomic expression, and it consists of a sequence of any characters other than parenthesis, comment characters, double quotes or blank-space

Here is an example list of atomic expressions separated by blank space:

*982 quick brown foxes jumped over the lazy dog.*

The sentence above consists of nine atomic expressions, with the ninth being the word dog and the trailing period character.

## String Literals

We can enclose a string in quotes if we wish to contain blank space characters as part of the element:

“982 quick brown foxes jumped over the lazy dog.”

The sentence above is a single string literal element. Parenthesis and comment characters are treated as part of the string.

“My name is Legion (not Lenny…) // just for the record”

The sentence above is again a single string literal element. If we wish to add double quotation characters to the string literal, we must prepend them with the escape character &:

“My name is Legion (not &”Lenny&”…)”

We can also encode blank space characters:

“My name is Legion&r&n&t (not &”Lenny&”…)”

In the sentence above &r is the ascii carriage return character, while &n is the ascii newline character and &t is the ascii tab character. Ampersand & was chosen as the escape character amongst a number of candidates as it was posited to cause the least number of conflicts specifying file paths in certain operating systems.

## Compound Expressions

When an expression is enclosed in parenthesis it is deemed a compound expression and only counts as one element in the sequence of elements in which it is defined, thus:

The quick brown fox (jumps over the lazy dog)

Is a sequence of five elements, with the fifth being a compound expression with five elements itself. The latter are referred to as child expressions and generally are indexed from zero, so that child 3 of the final expression is ‘lazy’. Note that the parenthesis characters do not require blank space separating them from their adjacent expressions.

## Root Expressions

Every S-Expression file has to start somewhere, and this is called the root expression. Root expressions are programmatically a type of compound expression, so as with all compound expressions they have children that are indexed from zero. In plain-text representation they are the only compound expressions that do not have parenthesis. They are also the only compound expressions without a parent.

## Null Expressions

If two parentheses have no characters between them other than blank space, they specify a null expression. Generally, most occurrences of null expressions are treated as errors and they have little use in the SexyScript programming language.

## Comments

There are two sorts of comments in Sexy S-Expression files, line comments and block comments. These work almost exactly as the specification for comments in C++.

Line comments begin with a double (forward) slash and mark a comment to the end of the line, in which the parser will skip parsing until the next line:

(Hello there!) // This is a line comment

Block comments begin with a slash and a star and end with a star and a slash, and mark a comment from the start to the end, where the parser will skip parsing until the final slash:

/\* This is a

Block comment

All the way to here: \*/

Certain S-Parsers can be made to retain comments for various uses, such as automatically documenting an API.

## Functions

A function is a named sequence of code elements that execute in turn. They may only be defined in a root expression. The syntax is:

(function <Name> <inputs> -> <outputs> : <body>)

Where:

<Name> is an atomic expression led by ascii uppercase A-Z and followed by any alphanumerics.

<inputs> is a sequence of compound expressions that define input types and input names.

<outputs> is a sequence of compound expressions that define output types and output names.

<body> is a sequence of program instructions that manipulate input and possibly assign values to the output. Note that both the input and the output elements are optional, but the atomic expressions -> and : are mandatory and require blank space either side of them. The -> element is called the mapping indicator, while : is called the body indicator. In mathematics a function is sometimes called a map, hence the terminology.

Example:

(function DoubleX (Float32 x)->(Float32 y): (y = 2 \* x))

Here we see a function for doubling the input and assigning the value to the output. In SexyScript functions do not have return values, they have outputs. Outputs default to some null value.

## Inputs

An input is a compound expression that specifies a type and a name for the input. Example:

(Float32 value)

Here ‘Float32’ is the type and ‘value’ is the name – or *identifier*. Type are alphanumeric sequences that begin with a capital letter, while *identifiers* are alphanumeric sequences that begin with a lower case letter. Enforcing case dependency in the language increases compile speeds, as the compiler can infer intent more easily, and it also reduces the permutations of style choices. Note that underscores and hyphens are not permitted in either types or identifiers, reducing style permutations to a minimum.

There are also other elements that may constrain the usage, such as const:

(const Vec3 position)

These are explained in the advanced documentation.

## Outputs

Very much similar to the inputs, an output is a compound expression that specifies a type and a name for the output. Outputs default to some null value, so for example, Float32 defaults to 0.0. As with inputs, types are specified in Pascal case (capital letters first) while values/identifiers are specified with camel case (lower case letters first).

## Primitive Types

All types are either primitive or derivative of a number of primitive types. The primitives consist of Int32, Int64, Float32, Float64 and Bool. Note that strings are not primitive types.

Int32 are 32-bit signed integers. There are no unsigned integers in the Sexy scripting language.

Int64s are 64-bit signed integers.

Float32 and Float64 are floating point numbers in the format of IEEE754, single-precision an double- precision respectively.

Bool are 32-bit integers, with the only two legal values being 0 and 1, with 0 indicating false and 1 indicating truth.

## Struct Types

Structs are memory compact sequences of primitive types and other derivative types that generally compatible with C++ structs.

They are defined in root expressions using the struct keyword:

(struct <name>

<fields>

)

Where <name> is the usual pascal-case type name to be used to specify the struct and <fields> is a sequence of compound expressions that define types are identifiers for each field:

For example:

(struct Vec4

(Float32 x)

(Float32 y)

(Float32 z)

(Float32 w)

)

This defines a struct of type Vec4 with four elements, x, y, z and w, each of which is a 32-bit IEEE754 floating point single-precision number.

One can also specify more than one identifier at a time for a given type in the same compound expression:

(struct Vec4

(Float32 x y z w)

)

This also defined Vec4 as a four-element structure. In C++ this element would be packed with 32-bit alignment, so no memory gaps appear between the elements.

Structs are evaluated recursively:

(struct Vec4

(Float32 x y z w)

)

(struct Matrix4x4

(Vec4 r0 r1 r2 r3)

)

This defines a 16 element matrix with four rows, each of which has four elements.

In a function body structure fields are accessed using the dot syntax:

(<struct-variable>.<field\_variable>)

If a field variable is itself of type struct, then it too can accept a dot to access its fields, ad infinitum:

(Matrix4x4 m)

(m.r0.x = 0)

Structure fields can be assigned all at once using the memberwise constructor semantic, that has the form (<struct\_variable> = <one-element-per-member>), where <one-element-per-member> is a sequence of expressions to assign to each field in order of declaration in the field definition.

Example with (struct Matrix4x4 (Vec4 r0 r1 r2 r3)) and (struct Vec4 (Float32 x y z w)) we can the four elements x, y, z w with 1, 2, 3 and 4 respectively be invoking the member-wise constructor for a Vec4:

(Matrix4x4 m)

(m.r0 = 1 2 3 4)

## Namespaces

Namespaces are sequences of pascal case words separated by the period character. The sequence of words to the left of a period give a parent namespace, while those to the right give the child. In the Example:

Sys.Type

Sys would be the parent, while Type is the child.

In this example:

Sys.Maths.F32

Sys is a parent of Maths.F32, and Sys.Maths is a parent of F32. When an expression specifies the full namespace, from the root the final component, it is called fully-qualified. This is sometimes abbreviated to FQ, where appropriate.

Namespaces are defined in the root expression using the following syntax: (namespace <name>)

Where <name> is the fully qualified namespace, from parent to final child.

## Aliasing

All functions and all types, primitive and otherwise can be given an alias in a namespace using the alias keyword

(alias <type> <namespace>.<alias-name>)

For example:

(struct Vec3 (Floa32 x y z w))

(alias Vec3 Sys.Geometry.Vec3)

These pair of expressions create a Vec3 type that are aliased in Sys.Geometry. They can be specified in functions using their alias. Aliases structures and functions may be accessed outside of the source files in which they are defined, so that aliasing makes such elements *public*.

## Using directives

In a root expression one can specify a number of *using directives* for a file. These are done with the syntax (using <namespace>) where <namespace> is a fully qualified namespace. The compiler will search these namespaces to identify aliases for functions or types when such names are not found in the local source file. (See aliasing). Using directives are there to reduce boiler plate in code and make it easier to read.

## Interfaces

Interfaces specify a group of functions that refer to a common object – called the *implementation*, or the *instance*. They are defined thus:

(interface <FQ-name>

<methods>

)

The <FQ-name> specifies the fully-qualified namespace followed by the interface name. Since the interface is specified with a namespace, it makes it a public entity. It is currently not possible to specify an interface without using a namespace as prefix.

<methods> is a sequence of compound expressions that specify what functions are defined for the interface. They are have the format:

(<name> <inputs> -> <outputs>)

The name is a pascal case function name, while <inputs> and <outputs> follow the same rules for functions. Unlike regular functions there is no body indicator character and no body of code.

## Classes

Classes exist to implement the functions specified by a number of interfaces, and exist for no other reason. Classes are intrinsically private entities; they cannot be aliased in a namespace and thus are inaccessible outside the source module in which each is defined. Functionality is projected out by using the interfaces that are public.

The syntax is:

(class <class-name> <implements>

<fields>

)

Here <class-name> is the Pascal case unique name for the class and <implements> is a sequence of compound expressions:

(implements <FQ-interface>)

Where <FQ-interface> specifies the interface that a class implements.

<fields> are the same specification that are found in struct – sequences of types along with identifiers.

Where a class is defined, the methods that the interface require also requires a definition. These are done at the root expression using the keyword method:

(method <class-name>.<method-name> <inputs> -> <outputs> : <body>)

Here the <class-name> and <method-name> are both pascal case and specify the function that is implemented. The remaining elements match those in the *function* specification.

Method names *Construct* and *Destruct* are reserved and explained below.

Example  
  
(interface Animals.IDog

(Bark (Float32 hz) ->)

)

(class Dog (implements Animals.IDog)

)

(method Dog.Bark (Float32 hz) -> :

(Audio.Play “woof.mp3” hz)

)

Here interface Animals.IDog specifies a function Bark that is implemented by class Dog.

In order for a class to compile every function in every interface specified by a class must be provided by a method expression in the class definition file.

## Constructors

Every class is created with a constructor method in the root expression that has the syntax

(method <class-name>.Construct <inputs> : <body)

As seen here, there are no outputs and no mapping indicator in a constructor. Constructors are called via factory calls. (Explained later).

Example  
  
(method Dog.Construct

## Destructors

Destructors are optional and are specified at the root expression:

(method <class-name.Destruct : body)

They are not permitted to specify either inputs or outputs. Destruct is called by the compiler when a class instance is about to be deleted. It gives a chance for the implementation to clean up resources. Destructors must not throw exceptions.

## Class Factories

To allow a source module to create instances of a class and access them through interfaces one must define a class factory in the module in which the class is defined. At the root expression use:

(factory <FQ-factory-name> <FQ-interface-name> <inputs> : <body>)

Where:

<FQ-factory-name> specifies a fully-qualified namespace, a period character and then as pascal case factory name. E.g Cars.NewVehicle.

<FQ-interface-name> specified a fully-qualified interface name, see *interfaces* for more information.

<inputs> has the same format for <inputs> in any *function*. The <body> consists of a series of program elements. Inside the <body> will be a construct directive:

(construct <class-name> <class-arguments>)

When a factory is called, the body is executed until a construct expression is reached. This creates an instance of the class on the heap, passes the <class-arguments> to the constructor (method <class-name>.Construct) and initializes the interface to refer to the class instance.

Since factories are aliased in a namespace they are public entities, and are thus the means to expose classes outside the modules in which they are defined.

Example:

(factory Animals.NewDog Animals.IDog (IString name): (construct Dog name))

## Invocation

Defining functions, methods and types does not tell us how they are actually used, merely what they are. For this we need to understand invocation.

## Invoking functions.

In the body of any function one can invoke any known function with the following syntax:

(<function-name> <input-args> -> <output-args>)

Where:

<function-name> specifies either the local function to be invoked or the aliased function. It can be prefixed with a namespace, and the compiler will attempt to use *using directives* to resolve it if it is not found locally.

<input-args> and <output-args> are a list of identifiers, each of the type that must match that specified in the invoked function definition.

Example:

(function DoubleX (Float32 x) -> (Float32 y): (y = 2 \* x))

(function QuadX (Float32 x) -> (Float32 y):

(Float32 x1)

(DoubleX x -> x1)

(DoubleX x1 -> y)

)

In the example DoubleX, specified twice within the definition of QuadX’s body, invokes the DoubleX function, first passing x and x1, assigning 2 \* x to x1, then secondly assigning 2 \* x1 to y. This has the effect of quadrupling x.

## 

## Invoking local variables

As in the example for QuadX, we see a typename and identifier (Float32 x1). This is the format of specifying a local variable. This creates a unique variable x1 of type IEEE457 single-precision floating point number. It is a temporary variable, created in a section of computer memory called *the stack*, which is erased once the compound expression in which it is invoked terminates. All variables are initialized to null values. For Float32 the value is 0.0.

Variables that are specified as input or output are part of a function definition – the compound expression representing the function, and are thus valid for the lifetime of a function. Lifetime is also known as *scope*. In Sexy *scope* is synonymous with being a descendant of a compound expression.

## Assignment to local variables from functions

We can invoke a local variable and at the same time assign it to the value of an output in the case that a function has only one output value using the return value grammar found in most languages:

(Float32 x = (DoubleX 2.0))

In this expression (DoubleX 2.0) is evaluated, and the first and only output Y is assigned to x at the creation of x.

## Get Accessors

If a function takes no inputs and has one output it gains the semantics of a *get-accessor*. These are functions that can be invoked without having to surround them with parenthesis in many circumstances.

(function PI -> (Float32 pi): (pi = 3.14159))

(function GetSin (Float32 x) -> (Float32 y): (Sys.Maths.F32.Sin x -> y))

(function Gubbins …:

(Float32 y = (GetSin PI))

)

Here, in the example above, PI is specified without containing parenthesis. Sin of Pi is 0, so y should be assigned with zero.

## Invoking Interface Variables

Interface variables are added to the stack within a function body by the syntax:

(<interface-name> <identifier>)

E.g

(IDog dog)

In that format they are initialized to the universal null-object specific to the interface. The null-object is a default implementation of the interface that implements null methods. A null method ignores all input and returns default null values on all outputs.

Interfaces can be initialized using the factory construction syntax:

(<interface-name> <identifier> (<factory-name <args>))

E.g

(IDog dog (NewDog “Rover”))

In this case the factory NewDog will construct a class instance and assign the dog interface to the instance. Instances are created on the memory heap. When all references to an instance are removed from the stack method Destruct is called on the class and the heap memory is deallocated.

## Interface Assignment from a Function

A function can specify an interface as an output, in which case the interface can be initialized as any other variable assignment:

Example:

(function CreateDog (IString name) -> (IDog dog):

(IDog newDog (NewDog name))

(dog = newDog)

)

(function Bark -> :

(IDog dog = (CreateDog name))

(dog.Say “woof”)

)

Above are two examples in one. We copy a reference of newDog to dog in the first function. As can be seen in the calling function Bark, variable dog is created on the stack, thus preserving the instance on the heap created within CreateDog. Once Bark returns, the last reference to dog is removed from the stack, and the dog instance on the heap is destructed.

## Invoking Methods

Once we have an interface variable, we are free to call any of its methods. We do so with this syntax:

(<interface-identifier>.<method-name> <inputs> -> <outputs>)

This is very much like a function invocation. Example:

(interface Animals.ICat

(Meow (Float32 dB)-> )

(MakeBiscuits -> )

)

(function Main (Int32 id) -> (Int32 exitCode):

(Animas.ICat cat)

(cat.Meow 130)

)

In our example, since cat was not constructed from an instance, it pointed to the universal null object and cat.Meow ignored input 130.0 and did nothing else.

## Method Get-Accessors

Just as with regular functions, a method that returns only one output and takes no input has get-accessor semantics and can often be used without enclosing parenthesis:

(interface Animals.ICat

(BiscuitCount -> (Int32 nBiscuits)-> )

)

(function Main (Int32 id) -> (Int32 exitCode):

(Animas.ICat cat)

(#printf “Biscuits:” cat.BiscuitCount)

)

## Exceptions

Sexy error handling is done through exceptions. This entails two parts, throwing and catching. Throwing is what is done when an error is detected and catching is what is done to handle it.

## Throwing

This is done in any function body via the syntax (throw <identifier>) where <identifier> is a variable of type Sys.Type.IException. Execution of the body terminates and control flow passes to the chain of callers until a catch handler is identified. For your convenience a function Sys.Throw exists. It has the format:

(Sys.Throw <error-number> <error-message>)

Where:

<error-number> is a system error code. This should be zero unless you understand how your execution environment handles the code and it is appropriate a non-zero value is used.

<error-message> contains an English language account of the error.

## Catching Exceptions

If no catch handler is found an exception will lead to script termination, which may lead to a debugger pop-up or programme termination depending on the execution environment.

Catch handlers have the syntax:

(try ( <try-body>)

catch <identifier> ( <catch-body> )

finally ( <final-body> )

)

Note the parenthesis around the <try-body> and <catch-body> expressions. <try-body> gives the sequence of compound expressions that define the flow of execution, these are as you will find in any function body. The <catch-body> is a sequence of compound expressions that will execute if any element in the <try-body> results in an exception being thrown. In this case <identifier> will refer to the exception instances passed in the (throw <identifier>) semantic.

The finally keyword and <final-body> are optional, and if present they define the sequence of compound expressions to execute either after a catch handler has completed, or the <try-body> was completed. So it acts to execute a code at the end of the exception irrespective of whether an exception was thrown. Note that if the catch handler throws an exception itself that leaves the catch handler, then the <final-body> is skipped.

## Nested Exception Handlers

It is perfectly legitimate for try…catch handlers to be placed inside other <try-body> elements. It is also legitimate for a <catch-body> to throw exceptions. The one place you must not throw an exception is in the Destruct method of a class.

## Primitive Algebra

Given x and y are two identifier names for the same primitive type, where the type is one of Float32, Float64, Int32 or Int64 then the following operations are supported:

(x + y)

(x – y)

(x \* y)

(x / y)

Note the blank space between the identifiers and the arithmetic operators. For these expressions it is legal to substitute x or y with numeric values:

(1 + x)

(x – 2.5)

(x \* 7)

(6 / 3)

However, the compiler needs to identify the meaning of the numeric value, and it does this by looking for assignment to a left-hand identifier:

(Float32 x = (6 / 3))

In the example above, the left-hand identifier is x, which is of type Float32. The compiler will then interpret the pair of numbers 6.0 and 3.0 each as 32-bit IEEE457 floats.

In the example (Float64 x = (6 / 3)), the numbers are interpreted as double precision literals.

Note, unlike some languages a binary expression (x OP y) will not compile without a value expected for an assignment or a function call.

So, (6 + 4) as a standalone statement is prohibited, while (y = (6 + 4)) is legal.

## Implicit Parentheses

The compiler is most efficient when the expression to the right of an assignment is a compound expression, as in:

(y = (2 + 4))

However, for the convenience of the programmer, the compiler will rewrite expressions internally, allowing the parenthesis to be omitted.

(y = 2 + 4)

It even works with function calls:

(y = DoubleX 2)

If you have chain of operators, the compiler will recursively compound expressions following the first operator onwards, so for example:

(y = 1 + 2 + 3 + 4 + 5)

gets converted to

(y = (1 + 2 + 3 + 4 + 5))

thence

(y = (1 + (2 + 3 + 4 + 5)))

thence

(y = (1 + (2 + (3 + 4 + 5))))

And finally:

(y = (1 + (2 + (3 + (4 + 5)))))

This allows the assignment to use binary operators to evaluate y to 10. This implicit conversion slightly slows the compiler, so if you have a large section of code where compilation time is critical you should add the parenthesis yourself. For most purposes the optimization is superfluous.

## Structures and arithmetic operators

In the expression (s = a OP b) where OP is one of +, -, \* or / and s is a struct types,

The compiler will attempt to invoke a method

<op-name><type-a><type-b>(<type-a> a)(<type-b> b)(<type-s> sum)

This is called operator *overloading*.

<op-name> is Add when OP is +

<op-name> is Subtract when OP is -

<op-name> is Multiply when OP is \*

<op-name> is Divide when OP is /

The <type-a> string is the local type name of a and <type-b> string is the local type name of type-b.

For example, if s, and b of type Sys.Maths.Vec2, then

(s = a \* b) will invoke (AddVec2fVec2f(Vec2 a)(Vec2 a)(Vec2 s))

The implementation of this function in Sys.Maths.F32 assigns the s vector from the sum of vectors a and b. Note that since structures are passed by reference operator overloading like this is very efficient, without having to create temporary objects as some languages would require.

If you specify *using directives* the compiler will attempt to use these to resolve the operator overloading function. If no resolution occurs the compiler may instruct you on how to specify the necessary function.

## Structures on the stack

You can create a structure on the stack using the syntax (<typename> <identifier>) where <typename> is the Pascal case struct name and <identifier> is the camel case variable name.

Fields can be initialized by a member-wise constructor:

(<typename> <identifier> = <args>)

Where <args> is the sequence of initial values, where one element matches each field in specified structure. If fields are themselves structures the fields can be recursively initialized in the argument list:

(Vec3 v = 1 0 1) // initialize the 3 components x, y and z to 1, 0 and 1 respectively.

(Matrix4x4 = (1 0 0 0)

(0 1 0 0)

(0 0 1 0)

(0 0 0 1)

) // member-wise construction of the four four-vectors that define the matrix fields.

Structures variables are passed to functions by reference:

(Vec3 a = 1 0 0)

(MultiplyVector a 3.0)

Here a function MultiplyVector takes a Vec3 reference and a value 3.0. We expect it to multiply the components of a by 3.0. Since struct references can be used to both read and write to a struct this makes structs useful for both input and output. This can dramatically improve the efficiency of many operations, particularly those on complex structs.

## Delta Operators

Float32, Float64, Int32 and Int64 variables can be modified using *delta-operators*. These have the syntax:

(a += b)

(a -= b)

(a \*= b)

(a /= b)

These are equivalent to

(a = (a + b))

(a = (a – b))

(a = (a \* b))

(a = (a / b)

Respectively.

The most common occurrence is the increment of an integer:

(i += 1)

## Control-Flow

Bodies of a function need code that can test for conditions and branch or loop, for this we have control-flow semantics.

## If Statements

Syntax is (if <condition> <body> else <other-body>).

Where <condition> is some boolean valued statement, such as a variable, method call that returns a single boolean valued output or a compound boolean valued expression.

<body> is a sequence of elements that are executed in the case <condition> evaluates to true and <other-body> is a sequence of elements that are executed in the case <condition> evaluates to false.

Example:

(Bool friendly = target.IsFriend)

(if friendly

(Say “Hello friend”)

else

(Say “Stay where you are!”)

)

## While Statements

Syntax is (while <condition> <body>)

Where <condition> is some boolean valued statement, such as a variable, method call that returns a single boolean valued output or a compound boolean valued expression.

<body> is a sequence of elements that are executed in the case <condition> evaluates to true

Once the <body> has completed execution returns to the top of the while loop, and this is repeated until the <condition> no longer evaluates to true.

Example:

(Int32 i = 0)

(while (i < 10)

(i += 1)

(#printf i “&n”)

) // Emits the sequence 1…10 with new lines after each iteration

## Do…while statements

Syntax is (do <body> while <condition>)

Similar to a while statement, but the body is executed before the condition is tested.

Example:

(do

(i += 1)

(#printf i “&t”)

while (i < 10)) // Emits the sequence 1…10 with tabs after each iteration.

## For Statements

Syntax is (for <initialization> <condition> <finalization> <body>)

Where

<initialization> is a compound statement to be executed before everything else.

<condition> is tested before each iteration of the loop

<finalization> occurs after each loop ends

<body> is executed only if the condition is true

For statements are similar to while loops, only they apply initialization and finalization to make writing certain loops easier.

Example:

(for (Int32 i = 0) (i < 10) (i += 1)

(#printf I “&n”)

) // This emits 0…9 with each iteration separated with a new line character

## continue

Within any kind of loop it is permissible to use the (continue) keyword. (continue) returns execution to the top of the loop, skipping any code beneath.

## break

All loops also permit use of (break) keyword. (break) terminates the loop early, skipping any code beneath. In *for* statements, finalization is also skipped.

## return

(return) in a function body terminates the execution of the function, returning control to the caller.

## goto / label

Labels can be defined using syntax (label <label-name>). Execution can be made to jump to any label by use of (goto <label-name>). Goto was introduced to the language to permit the creation of complex control-flow semantics via macros. Labels must be either in the parentage of the goto expression or belong to the same compound expression.

Example:

(while (i > 0)

(if (i = -1)

(goto unexpected)

)

)

(return)

(label unexpected)

(#printf “Well I did not expect it to be -1”)

## yield

In many environments SexyScript may be required to share its CPU resources with other algorithms running in the same thread – in this case one can use (yield) in any function body to return control to the C++ host program. The host can then resume execution of the script at later date. If done properly this system can allow the CPU to switch between millions of scripts on the same thread. Returning from a yield and resuming typically cost only a few nanoseconds of CPU time.

## finally

In addition to exceptions, *While* loops and *do..while* loops also allow for the keyword finally to be used. If ‘finally’ is added as an atomic expression to a loop body then all the succeeding compound expressions will be executed at the end of a loop iteration, whether the loop progresses to the finally keyword, or whether it is advanced to the end with a (continue) statement. This behaviour mimics semantics of the third element in a *(for …)* statements.

(Int32 i = 1)

(while (i <= 10)

(Sys.Print "Hello Mum!")

(continue)

(Sys.Print "Hello Dad!")

finally (i += 1)

)

The section of code in the example will print out “Hello Mum!” ten times. Without the *finally* section the loop would never end.

## Event Handling and Dynamic Programming

In the development of user-interfaces, or GUI programming, one frequently has the problem of developing systems of widgets. Widgets are small units of code that provide a quantum of GUI interaction, such as a scroll bar, an edit box, or a button. With there being a large number of events that can interact with a widget, such as mouse clicks, touch-screen events, keyboard focus changes, it can be cumbersome to handle all possible events for each widget, particularly when most are unhandled and should normally result in ignoring the event conditions. For this purpose dynamic dispatch was introduced into the language.

## (attribute dispatch)

Dynamic dispatch appears when an interface is defined and an (attribute dispatch) added into its compound expression, e.g:

(interface Sys.IEventHandler (attribute dispatch))

Here we define an interface that specifies no particular methods, but it is marked as a dispatch interface. On such an interface, the caller may attempt to call any method on it, providing it is a method with one input argument that must be a struct, and has no output arguments. Since structs are passed by mutable reference this allows any amount of information to be passed into our out of the method through the struct.

So for example we can try to invoke a mouse wheel event:

(struct MouseWheelEvent

(Bool wasHandled) // output

(Int32 nClicksVertical) // input

)

In some function body:

(MouseWheelEvent wheelEv)

(wheelEv.nClicksVertical = 2)

(Sys.IEventHandler handler = widget.GetEventHandler)

(handler.OnMouseWheelMoved wheelEv)

Even though the interface does not explicitly list OnMouseWheelMoved as a method, the dispatch attribute tells the compiler to try to forward on the method call to the implementing class, so as long as the class implements the method, i.e (className.OnMouseWheelMoved (MouseWheelEvent ev) -> ) the method will get invoked. In the event that the function name is not matched the method call is ignored. If the function name matches but the argument is different an exception is raised.

In the example we added a field (Bool wasHandled) to the MouseWheelEvent struct. Since Bool fields default to false on initialization, this value will be set to false. Thus one could establish a protocol where the caller detects whether the event is ignored by having all handlers assign true to *wasHandled*. The caller can assume that if wheelEv.wasHandled is false at the end of the invocation then the function was not matched. E.g:

(method Scrollbar.OnMouseWheelMoved (MouseWheelEvent wheelEv)-> :

(wheel.wasHandled = true)

)

# Boolean Valued Expressions

In the case of do…while loops, while loops and for loops the compiler will expect some expression to evaluate to a Bool value. Such expressions are also required when assigning a value to a Bool type variable:

(Bool isLossless = <…bool valued expression…>)

In such expressions a legal expression may be:

The atomic literal “true” – which evaluates to true. Example: (Bool isLossess = true)

The atomic literal “false” – which evaluates to false. Example: (Bool isLossess = false)

The variable name of any other Bool typed value.

A get-accessor that has a Boolean output value.

A compound expression that satisfies the laws of boolean valued compound expressions.

# Boolean Valued Compound Expressions

A function invocation that has one output of type Bool.

An expression of the form (not <Boolean-Valued-Expression>) which is false when <Boolean-Valued-Expression) evaluates to true and true otherwise. The keyword *not* is only valid in such expressions.

An expression of the form (<b1> <operation> <b2>) where <operation> is one of [and|or|xor]. In the case of *and*, the expression evaluates to true if and only if both b1 and b2 evaluate to true. In the case of *or*, the expression evaluates to true if and only if at least one of b1 and b2 evaluates to true. In the case of *xor*, the expression evaluates to true if and only if one and only one of b1 and b2 evaluates to true.

An expression of the form (<v1> <operation> <v2>) where v1 and v2 evaluate to an integer or floating point number of the same type and operation can be one of [>|<|>=|<=|!=|==]. The operators represent (respectively): greater than, less than, greater or equal to, less than or equal to, not equal and equal. The operators compare <v1> and <v2> and return true if the sense of the arithmetic expression is true, otherwise they return false.