Mu Programming Language

A Computer Graphics Extension Language
Edition 1.0
28 August 2019

Jim Hourihan

Table of Contents

1 Overview

Mu is principally targeted towards computer graphics applications. The original (usable) version appeared at Tweak Films around 2001-2002. Over the years its syntax and runtime has been refactored and evolved from a simple shading language to a nearly full featured general language. However, Mu is still probably best suited for computer graphics than other computing tasks.

The following discussion is mostly a result of experience in the feature film special effects industry (which I'll call film). One could argue that computing tasks for film are extreme in many cases and that achievements and trends there tend to trickle down into related CG disciplines (like games and television post production). Indeed, over a short period of time the distinction between these CG disciplines has blurred.

Mu is an attempt to unify a number of disparate uses of "scripting" and compiled languages in CG. Using Mu in conjunction with C++ applications tends to hit a sweet spot; since Mu is itself written in C++ attention has been paid to making it easy to embed and use in those applications. The static type system has a number of advantages that lead to "better" user code and often easier to maintain.

Some background on computing in the CG film industry is useful here. To start with, the big computational tasks which affect application language choice fall into these catagories:

1.1 Rendering

While there are a number of different kinds of rendering software, there are two that are used most in film production: renderers based on the Reyes algorithm (Pixar's RenderMan being the obvious example), and ray tracing (Mental Images' Mental Ray possibly producing the most ray-traced pixels on film to date.)[1] These two algorithms have very different profiles: the Reyes algorithm shines on a fast SIMD architecture with fast memory cache lines, and ray tracing (which typically results in a lot of point sampling) benefits mostly from raw speed. Typically render times follow the so-called *one hour rule*[2]: the rendering scene for a given shot will increased in complexity until the render time hits one hour at which point the complexity will no longer increase.

Its not atypical for a single frame to require gigabytes of image data (textures) and geometry. This is typically multiplied by the number of frames in a shot — which has unforunately *increased* over the last few years. With the current trend towards stereoscopic film and all CG productions (feature CG animation) this is becoming even more extreme. Tricks like locally caching data, wavelet compression of images, and procedural shaders are often used to reduce the data and throughput complexity. Some newer real world sampling techniques like BDRF, and real time motion and 3D geoemtry capture have futher increased data demands on renderers.

Its not surprising that production renderers have historically been written in C and more recently C++. These languages allow application programmers to very carefully manage computing resources. The cost of course is complexity of the code and long debugging

[1] Pixar's Photorealistic RenderMan has actually become a ray tracer as well, and Mental Ray is really a hybrid scanline renderer, but for purposes of this discussion I'll just resort to blatant over-simplification.

[2] It *used* to be one hour; now its pushing two to three hours.

periods. But usually its worth it: the renderer's efficiency can directly translate into time and money (and there is a very strong incentive to minimize these!).

Mu has been used for two purposes with regard to rendering: as a shading language (like the RenderMan shading language) and as scene generation and storage language. The first use was for scene generation as a particle instancing language.

1.2 Compositing

Compositing is similar to rendering: multiple images are rerendered into a new image. In the process pixels may be modified, moved, or generated. While not as compute intensive as 3D rendering, compositing can consume substantial resources as well.

Recently a trend has developed where renderers produce "partially" rendered images which are actually coefficents in the rendering equation; these are then recombined with new coefficients in the compositing software thereby granting more flexibility after rendering. For example, each light in a scene is effectively turned on by itself creating a layer which is then dialed up and down in the compositor (since light can be linearly combined). This has resulted in a much larger input image set than previously.

Compositing software is fairly complex compared to most software. Unlike renders, compositing software requires a good deal of user interface to make it usable.

Like a shading language, a high-level compositing language is typically SIMD. Mu has been used in this context to make a per-pixel compositing scripting language.

1.3 Animation and Modeling

Animation and modeling software can be separate, but the trend has been towards large swiss army knife feature sets which include *everything* you might want to do in animation and modeling combined. The software used for film (Alias' Maya for example) is the result of 20 years of evolution. These programs are huge and typically deal with complexities exceeding CAD software (in some uses they are the CAD software). It would not be far fetched to argue that some of these programs are the most complex software written to date.

Most require very workstation computers, fast graphics cards, fast network I/O and have massive amounts of user interface which is often procedurally generated. They are dumping grounds for advanced algorithms spanning computational geometry to computational physicals to signal processing.

Most of the current crop of 3D modeling and animation packages are "scriptable". This usually means they have a fairly high-level interpreted language availble for model and animation construction or even as an extension language for adding novel behavior. Usually the language is also used to control and define the user interface of the application (since this is typically a good design choice for large software projects). In addition to the above, Maya even uses the its language (MEL) as its scene file format.

Mu is ideally suited for these same tasks: its a very high level language with specific built-in types which are typically required for animation and modeling software. It can be used as an interpreted language or it can be compiled. The application can add opaque types and APIs to Mu which easily bind to internal functions. Since Mu was designed for perforance, the language does not become a huge bottleneck even during scene evaluation.

1.4 Simulation

Physical simulation has been a part of CG since its inception. Over time this has only increased. Simulation software for film has usually been embedded in *procedural animation* systems which allow the user to programmaticly control physical elements in time and space.

So to sumarize the computing landscape in CG film:

- **Large datasets are common**. Many applications will use one minus the number of bytes available to them.
- **Performance is really important**. The artist's attention span is closely related to rendering and simulation times.
- **Workers usually have informal training for programming**.
- **Workers are skilled tool users**.

2 Primitive Types

Mu types have two kinds of semantics: reference and value. Types with value semanitics are always copied when assigned or passed to a function as an argument. Basic number types all have value semantics.

2.1 Boolean type

Mu has a type for boolean values called `bool` and its two constants `true` and `false`.

Boolean values do not cast to other integral values by default as in the C language. So many C/C++ idioms are not applicable in Mu.

```
int a = 0;
if (a) doit();      // error
```

The above example does not work in Mu[1]. The `int` will not cast to a `bool`. The correct way to do the above is:

```
int a = 0;
if (a == 0) doit(); // ok
```

All of the conditional constructs take the `bool` type as their test argument.

2.2 `int64`, `int`, `short`, and `byte` types.

Mu has four integral value types. The `int64`, `int`, and `short` types are represented as signed twos complement. The `int64` type is 64 bits, the `int` type is 32 bits and the `short` type is 16 bits. `byte` is 8 bit unsigned. The binary representation in memory is machine dependant.

The integral types all obey basic arithmetic operations as well as the bitwise operators. None of the integral types cast to the boolean type `bool` automatically.

There is a distinction between the `byte` and `char` type in Mu. The `char` type is not assumed to be of any particular size. There are no "unsigned" integral types.

2.3 Floating point types.

The floating point type is called `float` and is a 32 bit IEEE floating point number. Mu can also be compiled with `half` (a 16 bit floating point number) as defined by ILM's Imath library. The usual arithmatic operators work on floats. Underflow and overflow should behave consistantly across platforms.

Mu floats do not throw exceptions by default.

[1] Well this isn't entirely true: you can make a cast operator from int and float to bool and get the C++ behavior

2.4 Character type.

The `char` type represents a unicode character.

To make a character constant in Mu, use signle quotes around a single character:

```
char c = 'x';
```

For unicode values when the input file is not encoded as UTF-8 or another accepted coding, you use the unicode escape sequence to specify non-ASCII characters:

```
char c = '\u3080';
```

Characters may be cast to strings.

```
string s = 'c'; // string => "c"
```

(See section on strings for more info). Operations on the `char` type are all related to distances between characters:

```
int diff = "c" - "a";   // int => 2
char a = 'a';
char b = a + 1;         // next character after 'a'
a++;                    // next character
assert(a == 'b');
```

2.5 The `void` Type.

The `void` type is useful in the context of function declarations. You cannot make a variable of type `void` since it has no value. The `void` type indicates that absence of a return type for a function definition:

```
function: print_it (void; int i) { print(i + "\n"); }
```

In the example the function `print_it` has no return value so the type `void` is used to indicate that.

2.6 The `nil` Type.

The value `nil` is used to set a reference variable to refer to "nothing". It is analogous to `NULL` in C/C++, `null` in Java, `nil` in lisp, or `None` in Python.

`nil` is of type `nil`. The type is special in that it can be automatically cast to any reference type.

You cannot test any type value against nil using `==`. Instead operator `eq` or `neq` must be used:

```
string s = ‘‘hello’’;
if (s eq nil) print("error\n"); // syntax error
```

2.7 Vector Types

The vector types are primitive types which are passed by value. To specify a vector type you use the `vector` type modifier:

```
vector float[4] hpoint = {1, 2, 3, 4};
vector float[3] point  = {1, 2, 3};
vector float[2] ndc    = {0.0, 1.0};
```

There are currently only three types which are allowed to be used with the `vector` type modifier. These types are: `float[2]`, `float[3]`, and `float[4]`.

Vector types allow access to their members using member variables. The member variables are called `x`, `y`, `z`, and `w`. In addition you can use the indexing notation (operator[]). For the two dimensional type, `z` and `w` are not available. For the three dimensional type, `w` is not available. For example, here is a normalize function:

```
function: normalize (vector float[3]; vector float[3] v)
{
    float len = math.sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
    return v / len;
}
```

You can also set the members directly:

```
vector float[3] v = {1, 2, 3};
v.x = v.y;
```

To make arrays of vectors, it may be necessary to include parentheses in the type definition:

```
(vector float[3])[] array_of_points;
```

See also symbolic assignment operator for creating type aliases.

2.8 Function Types

Function types are declared much like functions themselves. The type of a function encodes its return type and all its argument types (whether or not it has default values).

For example, this is the type for a function that takes two `int`s as arguments and returns an int:

```
(int;int,int)
```

Whitespace inside the declaration is ok, but for clairity it is omitted here. In practical usage, you might want to create a variable and assign a function to it like so:

```
function: add (int; int a, int b) { a + b; }
(int;int,int) x = add;
```

Since functions are first class objects in Mu, you can make arrays of them:

```
(int;int,int)[] int_funcs = { add };
```

For example, this makes a dynamic array of functions which return an int and take two ints and initializes it to have a single element: the function `add`.

Of all the type declaration syntax, the function type syntax is the most complex. Here's an example of a function type which takes a dynamic array of `string` and returns a function that returns an `int` and takes a `float[4,4]` matrix as an argument:

```
((int;float[4,4]);string[])
```

See the section on symbolic assignment for how to clean up a mess like the above.

2.8.1 The Ambiguous Function Type

One problem that arises in a type system with function overloading is ambiguity resolution (if ambiguity is allowed). When a function type expression is evaluated for an overloaded function, the result is the ambiguous function type:

```
mu> print
(;) => print
```

In this case, the `print` function is overloaded and therefor is returned as type `(;)`. An expression or variable of type `(;)` cannot be evaluated using `operator()`. However, the type will automatically cast to a callable function type (as seen in this interpreter session):

```
mu> (void;float)(print)
(void;float) => print (void; float)
```

In this case the result is no longer ambiguous. If you attempt to cast to a function type for which there is no overloaded instance, the cast will throw an exception.

3 Built-in Types

Mu has built-in syntax for various types of arrays, lists, maps, and strings.

3.1 Lists

Lists in Mu can be formed directly by enclosing a comma separated list of values in brakets:

```
[1, 2, 3, 4, 5]
```

Mu will infer the type of the list. The type of a list is expressed by enclosing a type name inside brackets. So the type of the above expression is `[int]` (a list of `int`). Mu does not have heterogeneous lists (without defining them yourself). So you can't do something list this:

```
[1, "two", 3.14]  // syntax error
```

There are three operators on lists and some syntactic sugar. The operators are `cons`, `head`, and `tail`.

3.1.1 `cons`

`cons` (['a]; 'a *`head`*, ['a] *`list`*) [Function]
: Returns a new list with *head* as the first element followed by each of the elements in *list*.

The cons operation can be made by either using the function `cons` or using the operator `:`. The first argument (or left operand) is any value. The second argument (or right operand) must be a list of the first argument type. So for a type `'a` as the first argument, the second argument must be `['a]`. For example the following are equivalent and result in the list `[1, 2, 3, 4]`.

```
1 : [2, 3, 4]
cos(1, [2, 3, 4])
```

3.1.2 `head` and `tail`

`head` ('a; ['a] *`list`*) [Function]
: Returns the first element in *list*.

`tail` (['a]; ['a] *`list`*) [Function]
: Returns the list of all elements in *list except* the first element in the same order they appear in the *list*.

The `head` and `tail` functions return the first element in a list (head) or the rest of the list (tail). `tail` will always return a list or `nil` if there is no tail.

```
head([1, 2, 3])     ⇒ 1
tail([1, 2, 3])     ⇒ [2, 3]
tail([1])           ⇒ nil
```

3.1.3 Pattern Matching with Lists

The list syntax can also be used to pull apart the values in a list by using it in a pattern:

```
let [x, y] = [1, 2];
assert(x == 1);
assert(y == 2);
```

Note that the number of elements in the list must match the number of elements in the pattern. If they do not match an exception will be thrown at runtime (or at compile time if can be detected).

The cons operator `:` can also be used in patterns. For example:

```
let h : t = [1, 2, 3]
```

The value of `h` will be `1`, the value of `t` will be the list `[2, 3]`. This is equivalent to doing the following:

```
let x = [1, 2, 3],
    h = head(x),
    t = tail(x);
```

The cons pattern will throw if the list on the right-hand-side has the value `nil`.

3.2 Tuples

Tuples are collections of a fixed number of heterogeneous types. Another way to think of them is as anonymous structures.

Tuples have a special syntax used for construction and destruction (pattern matching) which make them easy to use. To construct a tuple of two or more values, simple enclose them in parenthesis like so:

```
(1, "the number one")
```

Similarily, the type of the above expression is `(int,string)`. So tuple type declarations look similar to tuple values:

```
(int,string) x = (1, "the number one");
```

Tuple values (and tuple types) can be nested as well:

```
((int,string),float) x = ((1, "the number one"), 3.141529);
```

There is one exceptional circumstance with tuple values: if one of the elements is `nil`, the tuple type will be ill-defined and will produce a syntax error:

```
(1, nil)    // error
```

There is currently no way to make a *singleton* tuple value (a tuple of one element).

3.2.1 Indexing a Particular Element in a Tuple

Each element in a tuple is given a numerical name. The first element is called `_0` followed by `_1` followed by `_2`, and so on. For example to get the second element:

```
(1, "two")._0     ⇒ 1
(1, "two")._1     ⇒ "two"
```

The value of the first expression is `1` because the 0th element is begin indexed. It is not possible to dynamically index tuple elements because each element may have a unique type. The best way to think about a tuple is a struct with elements named `_0` through `_N` that you don't need to declare.

3.2.2 Pattern Matching with Tuples

Usually you don't bother with the type annotation and use pattern matching instead:

```
let x = (1, "the number one");
```

`x` in this example has type `(int,string)`. The tuple syntax can also be used when pattern matching to pull values out of the tuple (destruction):

```
let (a, b) = (1, "the number one");
```

So in this case `a` will have the value `1` and be of type `int` and `b` will have the value `"the number one"` and be of type `string`. Tuple patterns may also be nested:

```
let (a, (b, c)) = (1.234, (2, "three"));
```

The tuple pattern will throw if the matching tuple value is `nil`.

3.3 Dynamic Arrays

Mu has two different kinds of arrays: dynamic and fixed. Unlike C, the type specifier completely encodes the array type. For example:

```
float[] foo;
float foo[]; // syntax error
```

The first line makes a dynamic array of floats. The second line produces a syntax error in Mu because the C array syntax is not permitted. The dynamic array object has a number of member functions on it which can be called:

```
foo.size();         // returns number of elements in foo
foo.push_back(1.0); // appends the number 1.0 to the end of the array
foo.front();        // returns a reference to the first element in foo
foo.back();         // returns a reference to the last element in
foo.clear();        // sets the size to 0
foo.resize(10);     // resizes foo to 10 elements.
foo.rest();         // returns all but the first element as an array
foo[0];             // returns the 0th element
```

The array type declarations can be nested:

```
float[][] foo;              // array of array of floats
foo.push_back( float[]() ); // add a float[] to foo
foo[0].push_back(1.0);      // add a float to the 0th element of foo
```

Note that this is not the same as making a multidimensional array. You cannot currently make a multidimensional dynamic array. However you can make a multidimensional fixed array.

3.4 Fixed Size Arrays

Fixed arrays are similar to dynamic arrays, but the size is encoded in the type:

```
float[10] foo;      // make a 10 element fixed array of floats
```

The fixed array types have fewer operations than the dynamic array type (since they are a more restrictive type). However the basic array syntax is shared:

```
foo.size();         // returns size (ok)
foo.push_back(1.0); // syntax error! (no push_back function)
float x = foo[5];   // 5th element
foo[5] = x;
```

Like dynamic arrays, you can make arrays of arrays and mix them with dynamic arrays:

```
float[10][4] foo;   // 4 arrays of 10 floats
float[10][] foo;    // a dynamic array of 10 float fixed arrays
```

In addition, fixed arrays may have multiple dimensions. This is done by adding comma separated lists of sizes:

```
float[4,4] foo;     // a 4x4 matrix of floats
float x = foo[1,1]; // set x to 1st row 1st column value
foo[3,1] = x;       // copy x to the 3rd row 1st column value

float[4,4,4] foo;   // a rank-3 tensor
```

For graphics applications, its common to

3.5 Maps

3.6 Sets

3.7 String Type.

Strings conceptually contain a sequence of `char` type. However, the `string` type is currently independant of any sequence type. Strings in Mu are immutable; you cannot change the value of a string. Instead you make new strings by either concatenating existing strings or by using the formatting operator (`%`).

`string`s can be constructed as a constant from a sequence of characters in double or triple quotes. (See String Constants for more details).

```
string s = "Hello World";
string y = """A long string with
possible newlines embedded "quotes" and other
difficult characters""";
```

There are a few basic operations on strings:

hash size split substr

In addition, any value can be cast to a string (or rather a string can be constructed from any value).

3.7.1 Formatting

Strings can be formatted using the `%` operator. The left-hand-side of the operator is a format template string and the right side is a single value or a tuple of values that are substituted into the template much like C's printf function.

For example:

```
"%0.2f" % math.pi               ⇒ "3.14"
"%d" % 123                      ⇒ "123"
"%s and %d" % ("foo", 123)      ⇒ "foo and 123"
```

3.7.1.1 Formatting Directives

3.8 Regular Expressions and Syntax

3.9 Types Defined by Applications or Native Modules

Some applications or native modules may provide an opaque type. These are types that appear to have no structure from Mu, but have operators and functions which can operate on them. There is no way to define an opaque type in Mu, but a similar effect can be had by defining an enumeration as a union.

An example of an opaque type can be found in the `system` module. This module contains analogues to POSIX functions and types in Mu. One of the declared types `FILE` represents a standard C library file descriptor. This type cannot be deconstructed but can be operated on by `fopen`, `fclose`, `fread`, and `fwrite` among others. Opaque types often represent external resources like `FILE`.

4 User Defined Types: Structs, Classes, and Unions

4.1 Records

4.2 Object Oriented Features: Classes

4.3 Tagged Union Type

Mu has a type-safe data structure called a *union* which makes it possible to deal with values from more than one type. The implementation of a union in Mu is different than languages like C and C++ which give unrestricted access to values.

Unions declarations have the following syntax:

`union:` *`union-name`* `{` *`cnstr1 [ type1 ] [ | cnstr2 [ type2 ] ]*`* `}`

Here's a specific declaration:

```
union: Foo { A int | B float | C string }
```

The example declares a union called `Foo` which has three constructors: `A`, `B`, and `C`. Each constructor is separated by the `|` character. You could read the declaration as "`Foo` is an `int` which we'll call `A` *or* a `float` which we'll call `B` *or* a `string` which we'll call `C`". So, the constructor `A` when called as a function takes an `int`, `B` takes a `float`, and `C` takes a `string`.

From a theoretical perspective, a union can be thought of as a type which includes the values of other types plus a *tag* for each value. In order to retrieve the value of a union, the tag must be supplied which matches the tag on the value. In other words, to get a value from a union you need to know what its tag is. This type of union is called a *tagged union* as opposed to the C-like *untagged union* where only the value is stored.

In Mu, the `union` includes multiple *constuctors* (which are similar to class constructors) which are used as tags. Each constructor in the union must have a unique name. The constructor has a type associated with it; values from that type are stored in the union with the constructor tag. To make a `union` object you must call one of its constructors.

Continuing the above example of `Foo`, we can use the constructors to make instances of `Foo` like this:

```
Foo x = Foo.A(123);
Foo y = Foo.B(3.141529);
Foo z = Foo.C("hello");
```

Or to make it easier to get at the constructors `use` the type:

```
use Foo;
Foo x = A(123);
Foo y = B(3.141529);
Foo z = C("hello");
```

In any case, `x`, `y`, and `z` are all of type `Foo`. The arguments and overloading of the constructors *are the same as the underlying type's*. So, for example the `string` has constructors like these:

```
string(1) ⇒ "1"
string(1.234) ⇒ "1.234"
```

which means the `Foo.C()` can be called similarily because its underlying type is `string`:

```
Foo x = C(1); ⇒ x = Foo.C("1")
Foo y = C(1.234); ⇒ y = Foo.C("1.234")
```

So why not allow casting to unions? For example this seems like it would be ok:

```
Foo x = 1.123;   error  Can't cast to a union value
```

There are two reasons. The first is that constructor arguments for multiple constructors unions may be identical; in that case there is no obvious way to choose the proper one. The second is actually a feature of unions: you can declare multiple constructors that take the same type. So this is ok:

```
union Bar { A int | B int | C int }
use Bar;
Bar x = A(1);

let C q = x;   error  x's value was not constructed with C!
let A w = x;  ⇒ ok! w = 1
```

`Bar` has three constructors and the all take `int`. So the union values are really the values of the `int` type alone. However, the union constructor (tag) is still required to get the value. You could think of the union in this case making *flavors* of `int`.

4.3.1 Retrieving `union` values.

To retrieve a union value, you must use some form of pattern matching. There are no fields of a `union` as there are in the C languages. For example, to get the `int` out of the `Foo` union declared above:

```
Foo x = Foo.A(123);
int Foo.A i = x;    ⇒ i = 123
```

If the pattern does not match (which means the value in the union could not have been created with the pattern constructor) an exception will be thrown:

```
Foo x = Foo.A(123);
let Foo.B f = x;     error  throws: x will only match constructor A
```

This form of pattern matching used for `union` *deconstruction* is only used in cases where the type of the union's value is known ahead of time (so that the pattern will not fail). When type is not known, the `case` statement or expression is used instead. The `case` pattern syntax allows the use of union constructors like `let`.

```
Foo x = Foo.A(123);

case (x)
{
    A i -> { print("x's value is the int %d\n" % i); }
    B f -> { print("x's value is the float %f\n" % f); }
    C s -> { print("x's value is the string %s\n" % s); }
}

⊣ x's value is the int 123
```

Note that we didn't have a `use` statement in the last example: in a `case` statement where the value being matched against is a union, the union's type is automatically being *used* inside of it. So we don't have to prefix the constructor pattern with the union name.

4.3.2 Enumerations as Union Constructors

There is a special case of a `union` constructor which has the `void` type. To declare such a constructor simply omit its type. While the union may have no retrievable value of that constructor, it will still be *tagged* as such. Using this mechansim enumerated types can be created where the constructor **is** the value.

```
union: Weekday { Monday | Tuesday | Wednesday | Thrusday | Friday }
```

These constructors take no arguments, and there is an exception to the usual syntax for functions in the case of these constructors: no parenthesis are needed to call them.

```
Weekday yesterday = Weekday.Monday;

use Weekday;
Weekday today = Tuesday;
```

You cannot use `let` to match against enumerated unions. Only the `case` statement (or expression) can be used:

```
case (today)
{
    Monday      -> { ... }
    Tuesday     -> { ... }
    Wednesday   -> { ... }
    Thursday    -> { ... }
    Friday      -> { ... }
}
```

5 Functions and the Function Type.

Functions play a ubiquitous role in Mu. Constructors, operators, iteration constructs, and more are implemented as functions which can be overriden, passed as objects, or modified.

5.1 Function Declaration

Mu functions are declared using either the `function:` keyword (aka `\:`) or the `operator:` keyword. When binding a function to a symbol, the form is:

```
function: identifier signature body
```

The *identifier* can be any legal identifier (with some restrictions). The *signature* portion of the declaration looks like this:

```
( return-type ; type0 arg0, type1 arg1, type2 arg2, ... )
```

This object, a signature, indicates the return type and all of the arguments to the function. The first part of the signature, the *return-type* must always be present. The second part, the argument list, can be empty indicating that there are no arguments to the function. The last portion the *body* is a code block. Here's an example function that computes factorial:

```
function: factorial(int; int x)
{
    return x == if 1 then 1 else x * factorial(x-1);
}
```

5.2 Operator Declaration

Mu allows operator declaration and overloading. In Mu operators are just functions with syntactic sugar. The operator precedence cannot be changed, but any operator can be overloaded. The operator declaration syntax is similar to the function declaration syntax:

```
operator: operator-token signature body
```

The *operator-token* is a special sequence of characters that describes the operator. In most cases this is the same as the operator in use. For example, the operator `^` is not defined for floating point numbers. But you can make it into a power operator:

```
operator: ^ (float; float base, int power)
{
    float answer = base;

    for (int i=0; i < power - 1; i++)
    {
        answer *= base;
    }

    return answer;
}
```

Some operators are overloaded. In order to distinguish between the overloaded versions, the *operator-token* is different the literal operator token. For example the postfix increment

operator `var++` is different than the prefix increment operator `++var`. Each has a different special token as show below.

This is the current list of special operator tokens:

`_++` postfix increment

`_--` postfix decrement

`++_` prefix increment

`--_` prefix decrement

`?:` conditional expression (takes three arguments)

5.3 Function Overloading

Functions may be overloaded. In other words, multiple declarations of the same function may be made as long as their arguments differ in type or number. For example:

```
function: area (float; triangle t) { ... }
function: area (float; circle t) { ... }
function: area (float; rectangle t) { ... }
function: area (float; shape t) { ... }
```

When the area function is applied to a circle, Mu will choose the appropriate version of the area function. In the above example, assuming that triangle, circle, and rectangle are all derived from the type shape, you can apply the area function to any other object that is derived from shape as well (Mu will choose the last area function above). When a direct match for an overloaded function is not found, Mu will attempt to use casting rules to make the function call.

5.4 Default values

Functions may have default values as long as every parameter after a parameter with a default value also has a default value. (This is the same rule that C++ has concerning default parameter values.)

```
function: root (float; float a, float b = 2)
{
    return math.pow(a, 1.0 / b);
}
```

The function "root" can be called either like this:

```
root(2.0, 2.0);
```

or this:

```
    root(2.0);
```

either way will return the value of math.pow(2, 0.5). It is an error to declare a function like hits:

```
function: root (float; float a = 2, float b)
{
    // ...
}
```

In this case parameter "b" follows parameter "a" which has a default value. Because "a" was declared with a default value, "b" must also be declared with a default value.

5.5 Returning from a Function.

There are two ways to return a value from a function. The first way is identical to C/C++: use the `return` statement:

```
return return-expression
```

The *return-expression* must be the same as the function return type or something that can be cast to it automatically. The `return` statement may appear anywhere inside the function. There can be multiple `return`s from a function.

Alternately, since blocks of code are also expressions, you can omit the `return` statement.

```
function: add (int; int a, int b)
{
    a + b;
}
```

This is not a syntax error because the last statement of the code block returns an `int` and therefor becomes the return value of the function. This is unlike C/C++ which require the `return` statement exist unless the function returns `void`.

5.6 Unnamed (Anonymous) Functions

An anonymous function can be created by omitting the function name in a function definition. For example, the good old `add` function as an anonymous function looks like this when fed to the interpreter:

```
mu> function: (int; int a, int b) { a + b; }
⊣ (int;int,int) ⇒ __lambda (int; int a, int b) (+ __lambda.a __lambda.b)
```

The result is a function object. You can assign this value to a variable if you like and call the function through it.

```
mu> global let x = function: (int; int a, int b) { a + b }
⊣ (int;int,int)& ⇒ __lambda (int; int a, int b) (+ __lambda.a __lambda.b)
mu> x(1,2)
⊣ int ⇒ 3
```

Unambiguous function objects all have `operator()` defined for them. So you can call the function either through a variable or an expression that returns a function.

6 Constants and Initialization

6.1 Integral Type Constants

hex octal decimal int64 v int

6.2 Floating Point Constants

engineering notation

6.3 String Constant Syntax

single Quotes. unicode escapes.

6.4 String Constant Syntax

Double quotes. Unicode escapes. Control escapes. Triple quotes. String constant juxtaposition. Handling of newlines in strings.

6.5 USer and Built-in Type Constants

The reference types which have constructors can be initialized using curly brackets when assigned to a type annotated variable. This applies to all types not just collections. However, this syntax is most often used with collections.

Using arrays as an example:

```
float[4] foo = {1.0, 2.0, 3.0, 4.0};
```

Multidimensional arrays:

```
float[2,2] foo = {1.0, 2.0, 3.0, 4.0};
```

also have the form of single dimensional arrays. However arrays of arrays:

```
float[2][2] foo = { {1.0, 2.0}, {3.0, 4.0} };
```

use nested brackets. Similarily dynamic arrays can be initialized:

```
float[] foo = {1, 2, 4, 5, 6, 7};
```

trailing commas are accepted. You may also use non-constant values in an array construction:

```
float a = 5.0;
float[] foo = {a, a*2, a*4};
```

6.5.1 Constants

In addition, you can make a constant by putting the type in front of the braces. For example if there is a function `bar()` that takes a dynamic array of ints, you can supply a constant to it like this:

```
bar( int[] {1, 2, 3, 4} );
```

6.5.2 Use with Patterns

The initializer syntax will not work with patterns because the type of the expression is underconstrained:

```
let a = {1, 2, 3 }; // error
```

The left-hand-side of assignment is a pattern, and the right hand side has an unknown type. In this case it could be a list, a dynamic array, a fixed array or a struct or class. So the type of `a` is ill-defined. If in this case we meant `a` to be of type `int[]` for example, we could use the constant syntax and do the following:

```
let a = int[] {1, 2, 3};  // ok
```

This is well defined.

6.6 Converting Basic Constants

Mu recognizes some basic constant suffixes. More can be added. Suffixes are defined in the `suffix` module as normal functions. For example the function `suffix.f` is declared like this:

```
module: suffix
{
    function: f (float; int i) { i; }
    function: f (float; int64 i) { i; }
}
```

In use, the suffix would appear *after* a integral numeric constant such as `123f`. The `f` suffix function is called by the parser to make a constant float out of the token `123`. Of course you could also just write it `123.0` which would have the same affect.

A more useful example would be a regular expression suffix:

```
module: suffix
{
    function: r (regex; string s) { s; }
}
```

The function `r` could be used to create a list of regular expressions like so:

```
["foo.*"r, "bar.*"r, "[0-9]+"r]
```

As opposed to this:

```
[regex] {"foo.*", "bar.*", "[0-9]+" }
```

Or as a way to diambiguate overloaded functions:

```
\: foo (void; regex a) { ... }
\: foo (void; string a) { ... }

foo("[0-9]+"r);   // calls first version
foo("[0-9]+");   // calls second version
```

For regular expressions, this has the added benefit of forcing the compilation of the regular expression during parsing instead of at runtime.

A more (possibly sinister) usage of suffixes is to represent mathematical or physical constants (or units) using them:

```
module: suffix
```

```
{
    function: pi (float; float x) { x * math.constants.pi; }
    function: pi (float; int x) { x * math.constants.pi; }
    function: i (complex float; float x) { complex float(0,x); }
}

let c = 6 + 12i;    // alternate complex number constant form!
let twoPI = 2 pi;   // questionable use of suffix!
```

6.7 Anonymous Function Constants

An anonymous function can be created by omitting the function name in a function definition.

For example, the good old `add` function as an anonymous function looks like this when fed to the interpreter:

```
mu> function: (int; int a, int b) { a + b; }
(int;int,int) => __lambda (int; int a, int b) (+ __lambda.a __lambda.b)
```

The result is a function object. You can assign this value to a variable if you like and call the function through it.

```
mu> global let x = function: (int; int a, int b) { a + b }
(int;int,int)& => __lambda (int; int a, int b) (+ __lambda.a __lambda.b)
mu> x(1,2)
int => 3
```

Unambiguous function objects all have `operator()` defined for them. So you can call the function either through a variable or an expression that returns a function.

7 Polymorphic and Parameterized Types

7.1 Type Variables

7.2 Typeclasses: Families of Types.

8 Variables

Mu types have two kinds of semantics: reference and value. Types with value semanitics are always copied when assigned or passed to a function as an argument. Basic number types all have value semantics.

8.1 Reference Type Construction Semantics

Reference types are allocated by the programmer or the runtime environment. For the programmer, this is done by calling one of the constructors for the type. For example, you can create a string object like this:

A variable that holds a reference types is actually better defined as a variant type; the value can be a member of the storage type or it can be nil.

```
string foo = string(10);
string bar = string();
```

which creates the string "10" from the integer 10 for the variable foo and the default empty string for the variable bar. When a variable is declared without an initializer, Mu will supply the default initializer or nil if there is none:

```
string baz;     // these are the same
string baz = string();  //
```

If you explicitly want to set an object type variable to nil, you should do that when intializing it:

```
string foo = nil;     // no object is constructed
```

This is not true of arrays however:

```
string[] bar;     // creates string[] object
bar.resize(1);
bar[0].size();     // whoops! that element is nil!
bar[0] = string();     // no longer nil
```

Because objects are passed by reference, setting one variable to another can have interesting results:

```
string foo = "ABC";
string bar = foo;
bar += "DEF";
print( foo + "\n" );
```

The output of the above example is "ABCDEF". Because the variables foo and bar are references to the same underlying string object, mutating operations through the bar variable will be visible through the foo variable. You can test for this condition using the operator `eq`:

```
if (foo eq bar) { print("they're the same!\n"); }
```

If you want to prevent this behavior, you need to make a copy:

```
string foo = "ABC";
string bar = string(foo);
bar += "DEF";
print( foo + "\n" );
```

In this case the string "ABC" will be printed because foo is referencing a different object than bar.

8.2 Kinds of Variables.

There are three kinds of "variables" in a Mu: global, local, and fields.

Global variables can be declared in any scope by preceding the variable declaration with the `global` keyword. There is a single location in the Mu process which represents a global variable and its lifetime is the same as the process.

The scope of the global variable's symbol declaration determines how and where the global variable can be accessed just like any other symbol. For example to make a global variable that is also globally accessable requires declaring in the top-most scope or a scope accessable from the top-most scope. For example all of the following are global variables that are accessable from any part of a Mu program:

```
global int x;
module: amodule { global int y; }

function: foo (void;) { print(x); } // ok
function: bar (void;) { print(amodule.y); } // ok
```

However, if the variable is declared in a function scope, then it is only accessable by parts of the code which can access symbols at the same scope:

```
function: foo (void;) { global int x; }
print(x);   // error! can't access it

function: bar (void;)
{
    global int x;
    function: foo (void;) { print(x); }   // ok
    print(x); // ok
}
```

Local variables exist on the program stack and exist only while the declaration scope is active. In the case of a function, this is while the function is executing. The variable has a unique copy for each function invocation just like a function's parameters.

Local variables are accessable on in the scope in which they are declared:

```
int x;
print(x);   // ok
function: foo (void;) { print(x); }   // error
function: bar (void;)
{
    int x;
    print(x); // ok
}
```

Fields are variables that are part of a larger object. For example, the two dimensional vector type has three field variables called "x" and "y". These variables are addressable using the scope (dot) notation:

```
vector float[2] v;
v.x = 1;
v.y = 2;
```

The lifetime of a field is the same as the lifetime of the object (its scope).

8.3 Variable Declarations

Mu is a statically typed language. It currently does very primitive inferencing and therefor requires type annotation for variables in many cases. Variable symbols are assigned a type which is immutable – this is called its storage type. The variable will only ever hold values of its storage type.

The annotated variable declaration statement has the one of following forms:

```
type variable [, variable ...]
type variable = expression [, variable = expression, ...]
```

In usage this looks like this:

```
int x;
int y = 1;
int z = 1, q = 10;
```

9 Pattern Matching

Besides the usual imperative variable assignment syntax, the `let` statement can be used to match patterns. There are two benefits to using let with pattern matching: multiple nested assignments can be made simultaneously and the types of the variables can be figured out by the parser so you don't have to annotate them.

Similarily, the case statement and expression can match patterns and values while assigning symbols to values without type annotation.

The ability of the parser to figure out types of symbols with annotation is called type inference. This is a common feature of functional languages in the ML family (O'CaML and Haslkell for example). Mu currently uses a very restricted type inference algorithm.

9.1 Assigning Variables with Patterns

The `let` statement defines symbols in the current and nested scopes. Symbols assigned using `let` are immutable (the values cannot be changed). The general form of the statement is:

```
let let-pattern1 = expression1
    [, let-pattern2 = expression2, ...] ;
```

The *let-pattern* can be any of the following:

- A single symbol which is assigned the value of its expression. This is the simplest usage of `let` and will never result in an exception being thrown. For example `let x = 0` assigns the type `int` to the symbol `x` and causes the value `0` to be bound to it.
- A tuple destructor pattern which binds the pattern symbols to each of the elements of a tuple expression. You can only use the tuple pattern if the bound expression is a tuple type. In addition the number of elements in the pattern must match the number of elements in the expression tuple type.

  ```
  let (a, b, c) = (1, 2, 3);
      ⇒ a = 1, b = 2, c = 3
  ```

- A list destructor pattern which bind the pattern symbols to each of the elements of a list expression. Like the tuple pattern, the list pattern requires that the number of symbols in the pattern be equal to the number of elements in the list value at runtime. If the number of elements does not match at runtime or the parser can figure out that the number does not match a parse time, an exception will be thrown.

```
let x = [1, 2, 3]; // without list pattern
    ⇒ x = [1, 2, 3]

let [a, b, c] = [1, 2, 3];
    ⇒ a = 1, b = 2, c = 3

let [d, e] = [6, 7, 8];
    error  num symbols != num elements
```

- A cons pattern which pulls apart the head and tail of a list. This pattern, like the list destructor pattern, requires a list expression. Two symbols are supplied: the head and tail symbol:

```
let h : t = [1, 2, 3, 4];
    ⇒ h = 1, t = [2, 3, 4]

let h0 : h1 : t = [1, 2, 3];
    ⇒ h0 = 1, h1 = 2, t = [3]
```

- The structure destructor pattern which pulls apart structure fields. This pattern can also be used on tuples and lists, but cannot be used on arrays.

```
struct: Foo { int a; float b; }
let {a, b} = Foo(1, 2.0);
    ⇒ a = 1, b = 2.0
```

- A union type constructor pattern. The expression supplied must be of the union type. If the constructor does not match the value constructor type, an exception will be thrown:

```
union: Bar { A int | B float }
let Bar.A {x} = Bar.A(1);
    ⇒ x = 1

let Bar.B {x} = Bar.A(1)
    error  Expression is of type Bar.A
```

- Finally, the symbol _ (underscore) can be bound to any expression. The expression is evaluated at runtime (if it is not pure), but the value is ignored.

Each of these patterns can be combined with the others and nested to arbitrary depths. This makes it easy to pull values from inside nested structs, tuples, and lists easily.

```
struct: Foo { int a; [int] b; }
let ([a, b, c], d, {e, f : g}) = ([1,2,3], 4, Foo(5,[6,7,8]));
    ⇒ a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = [7, 8]

let (a, b, _) = (1, 2, [3,4,5,6]);
    ⇒ a = 1, b = 2
```

9.2 Pattern Matching to Control Flow

9.3 Case an an Expression

10 Flow Control

Mu has the usual cadre of flow control statements for a C-family language plus a few additions.

All of the flow control statements are actually functions and they all return a value which depends on the statements they execute. Usually the return value is only useful in the context of a function definition where the last statement in the function is one of the flow control constructs.

10.1 Conditional Execution, the `if` Statement

The `if` statement has two forms: one with an `else` clause and one without. The test expression for an `if` statement must be of type `bool`. The general forms are:

```
if ( bool-expression ) if-true-statement
if ( bool-expression ) if-true-statement else if-false-statement
```

The return value of the `if` construct function is the return value of the *if-true-statement* or the *if-false-statement*. When there is an *if-false-statement* present, it must return the same type as the *if-true-statement*.

Because the integral and float types in Mu do not cast to type `bool` by default, many C/C++ idioms involving the if statement do not work:

```
int x = 1;
if (x) doit();          // error, x is not bool
if (x == 1) doit();     // ok
```

Some more examples:

```
if (x == 1)
{
    doit();
}
else
{
    stopit();
}
```

Also note that the C++ ability to declare variables in the test expression are not valid in Mu:

```
if (bool foo = testfunc()) doit();  // error
```

Since the test expression must be of type `bool`, the C++ism would not be very useful in Mu.

10.2 `case` Statements.

The general form is:

```
case ( value-expression )
{
    case-pattern0 -> { statement-block0 }
    case-pattern1 -> { statement-block1 }
    case-pattern2 -> { statement-block2 }
    ...
}
```

Unlike the `if` statement, the `case` statement can match over any value type that can appear in a pattern.

The first matching pattern will cause the corresponding statement to be executed. A simple example matching values might be:

```
string s = ...;

case (s)
{
    "one"     -> { print(1); }
    "two"     -> { print(2); }
    "three"   -> { print(3); }
    x         -> { print("Don't know what %s is\n" % x); }
}
```

10.3 Conditional Expressions; A Variation on `if`

The condition expressions in Mu looks similar to the conditional statements.

10.3.1 `if-then-else` Expression

In the case of the `if` conditional expression the keywords `then` and `else` are required. No parenthesis or brackets are required:

```
if test-expression then if-true-expression else if-false-expression
```

The *if-true-expression* is only evaluated if *test-expression* evualuates to `true`. Otherwise, the *if-false-expression* is evaluated. In either case, the value and type of the conditional expression is the same as the evaluated expression.

The type of the *if-true-expression* and the *if-false-expression* must be the same or case to the type. If the types differ, the parser will attempt to find the least lossy type cast that will make them match.

The conditional expression can be used anywhere any other expression can be used, but because of its precedence the entire expression may require parenthesis':

```
let x = if somecondition() then 0.0 else 3.1415;
print(if x == 3.1415 then ‘‘its pi’’ else ‘‘its not pi’’);
let y = x + (if x > 0.0 then 1.0 else -1.0);
```

10.3.2 `case` Expression

```
case test-expression of pattern1 -> expression1
                        pattern2 -> expression2
                        pattern3 -> expression3
                        ...
```

10.4 Fixed Number of Iterations Looping.

The `repeat` statement has the general form:

```
repeat ( int-expression ) statement
```

The *int-expression* is evaluated once. *Statement* is then evaluated that many times.

```
// outputs "hello hello hello "
repeat (3) print("hello ");
```

10.5 Simple Looping

The `while` and `do-while` constructs have the form:

```
while ( test-expression ) statement
do statement while ( test-expression ) ;
```

The *test-expression* must be of type `bool`. It is evaluted either before *statement* (or after in the case with `do-while`). If *test-expression* is `true`, then *statement* will be evaluated.

In the cast of `do-while`, the *statement* is always evaluated once since the test expression is evaluated after.

10.6 Generalized `for` Loop

The `for` construct in Mu is similar to the C/C++ statement of the same name:

```
for ( declaration-expr; test-expr; tail-expr ) statement
```

The *declaration-expr*, *test-expr*, and *tail-expr* are all optional. If the *test-expr* does not exist, it is as if the *test-expr* where `true` (the loop never terminates).

Test-expr must be of type `bool`.

As in C++, the *declaration-expr* can declare one or more variables:

```
// outputs: "0 1 2 "
for (int i = 0; i < 3; i++) print(i + " ");
```

In this case the variable `i` is in the scope of `statement` and does not appear outside the loop:

```
// outputs: "0 1 2 "
for (int i = 0; i < 3; i++) print(i + " ");

print(i + "\n"); // error "i" not defined here.
```

10.7 Operating on Every Collection Element

The `for_each` construct has the general form:

```
for_each ( identifier ; collection ) statement
```

Currently, *collection* can only be a dynamic or fixed array or a list. The *identifier* is bound to each element of *collection* then the *statement* is evaluated.

```
//  Outputs: "1 2 3 "
int[] nums = {1, 2, 3};
for_each (x; nums) print(x + " ");
```

10.8 Iteration on Collections over Indices

The `for_index` construct has the general form:

```
for_index ( identifier [, idenitifer, ...] ; collection ) statement
```

Currently, *collection* can be a dynamic or fixed array. The *identifier* is bound to each element of *collection* then the *statement* is evaluated. The number of identifiers must be the dimension of the collection.

```
//  Outputs: "1 2 3 "
int[] nums = {1, 2, 3};
for_index (i; nums) print(nums[i] + " ");
```

For multidimensional arrays:

```
//  Transpose a matrix
float[4,4] M = ...;
float[4,4] T;
for_index (i, j; M) T[i,j] = M[j,i];
```

10.9 Break and Continue: Short Circuiting Loops.

All of the looping constructs can be "short-circuited" using `break` and `continue`. These function just like their C/C++ conterparts. `break` and `continue` will terminate the inner most loop.

```
while (true)
{
    string x = doit();
    if (x == "stop it") break;
}
```

In the example above, the `while` loop will never terminate if `doit()` does not return "stop it".

If the loop as a *test-expression*, then `continue` can be used to cause the flow of control to skip the rest of the *statement* being iterated. In the case of a `for` loop, this will cause execution of the *tail-expr* before the *test-expr*.

```
// outputs: "1 2 "
for (int i=0; i < 10; i++)
{
    if (i > 2) continue;
    print(i + " ");
}
```

10.10 The `throw` statement

The `throw` statement, which raises and exception, has two forms:

```
throw throw-expression
throw
```

throw-expression can be of any reference type. So you cannot throw an `int` or `float` for example because these types are value types.

The second form is applicable only inside a the catch clause of a `try-catch` statement. In that context, it rethrows the last thrown value.

10.11 The `try-catch` statement

The `try-catch` form is:

```
try
{
    try-statements
}
catch ( catch-expression-1 )
{
    catch-statements
}
catch ( catch-expression-2 )
{
    catch-statements
}
...
catch ( catch-expression-N )
{
    catch-statements
}
catch (...)
{
    default-catch-statements
}
```

The `try` section and one of the `catch` sections are mandatory. The use of a default catch statement — one whose *catch-expression* is `...` — is optional.

The statements in *try-statements* are evaluated. If during the course of evaluation an exception is raised by a `throw` statement, control will be returned to the `try-catch` statement. At that point a type match is attempted between the object thrown and each of the `catch` clauses of the `try-catch` statement. The first `catch` clause that matches has its *catch-statements* evaluated. The default `catch` statement — which has `...` as its *catch-expression* will catch any type.

An example best explains it:

```
try
{
    throw "no good!";
}
catch (string s)
{
    print("caught " + s + "\n");
}
```

In this case the value "no good!" will be caught by the catch clause. The *catch-expression* in this case declares a varaible which is then assigned the thrown value. In the case of the default catch, you do not have access to the value:

```
try
{
    throw "no good!";
}
catch (...)
{
    print("caught something!");
}
```

There is no way to identify the caught type, but clean up can occur. If you need to rethrow the value you can use `throw` with no arguments:

```
try
{
    throw "no good!";
}
catch (...)
{
    print("caught something! rethrowing...");
    throw;
}
```

This works for any type of catch clause. In addition, you can throw a completely different object if you need to:

```
try
{
    throw "no good!";
}
catch (...)
{
    print("caught something!");
    throw "something else";
}
```

If no catch clause matches the thrown object type, its equivalent to a default catch that simply rethrows:

```
try
{
    something_that_throws();
}
catch (...)
{
    throw;
}
```

10.12 The `assert()` Function.

Although the `assert()` function is not strictly part of Mu's exception handling, the built-in function does provide a helpful way to debug run-time problems.

`assert` (*void; bool* `testexpr`) [Function]
: *testexpr* is evaluated. If the value is true, nothing happens. If the value is false, an exception is raised. The exception object will contain a string representation of the *testexpr* which can be output.

For example the `mu` interpreter when present with this:

```
int x = 1;
assert(x == 2);
```

will produce this:

```
ERROR: Uncaught Exception: Assertion failed: (== x 2).
```

Currently, assert will produce a lisp-like expression indicating the failed *testexpr*. Note that if partial evaluation of the *testexpr* occurs, you may get a surprising result:

```
mu> assert(1 == 2);
ERROR: Uncaught Exception: Assertion failed: false.
```

In the above case the *testexpr* was partially evaluated to false early in the compilation process.

11 Namespace (Scoping) Rules

11.1 How Symbols are Assigned to Namespaces

When a symbol is declared (like a variable or a function) it is declared in an existing *namespace*. A namespace is itself a symbol. For example functions, types, and modules are all namespaces. The namespaces form a hierarchy; the root is called the *global namespace*. The global namespace has no name, but is pointed to by the global symbol `__root` (which is of course in the global namespace!). Normally you don't need to access the global namespace by name.

Since every namespace is in the global namespace, we can form absolute or relative paths to symbols. This makes it possible to disambiguate two symbols with the same name, but that live in different namespaces:

```
module: Foo
{
   global int i;
}

function: bar (void;)
{
    int i = 10;
    print("%d\n" % (i + Foo.i));
}
```

In this example, in function `bar` we need to add two symbols named `i`. In order to indicate which one we are refering to the path to the global variable `Foo.i` is specifically given.

11.2 Declaration Scopes and Rules

Each of the following is a namespace in which symbols can be declared:

- Modules. Modules are namespaces that can be accessed from any other namespace. Nested modules can access functions, variables, and types declared in the current module namespace or parent module namespaces without using a path to the symbol.
- Functions. Local variables, functions, modules, etc, that are declared inside a function body are only visible in the scope of that function or its child namespaces. It is not possible to reference a symbol inside a function (including its parameters) from outside of the function.
- Types. Fields of types are accessible to outside namespaces though the dot notation of objects. Global variables, functions, and types declared in the scope of a type are accessible from any namespace. Nested types follow the same scoping rules as nested modules (and can be intermixed with modules as well). This includes both record-like types (`struct` and `class`) as well as the `union` type.

11.3 Loading a Module

The `require` statement makes sure that a named module has been loaded. Code following a `require` statement is guaranteed to find symbols refered to in the specified module.

```
require module_name
```

11.4 Using a namespace

The `use` statement makes any accessible namespace visible *to the current scope*.

```
use namespace
```

Once the current scope ends, the namespace referenced by `use` is no longer visible. This is a convenience to make code less verbose:

```
module: Foo
{
   function: add (int; int a, int b) { a + b; }
}

use Foo;
add(1,2); ⇒ 3
```

In this case, the symbols in module `Foo` become visible to the current scope. So the function `add` can be directly refered to without calling it `Foo.add`. This also works for types:

```
class: Bar
{
    class: Thing { ... }
}

use Bar;
Thing x = ...;
```

Here the type `Thing` is directly visible becase `Bar` is being used. This also applies to any functions declared in `Bar`.

11.4.1 Using a Module Implies `require`

The `use` statement not only makes a namespace visible, but can find namespaces on the filesystem before doing so. In other words, `use` can also perform the same function as `require`:

```
use io;
fstream file = fstream("x.tif");
```

Here the `io` module may be loaded if it was not already required by some other part of the program. `use` will attempt to load a module if it cannot find any namespace with the same name as the argument. When applied to modules, `use` can be thought of as doing two things:

```
require module_name
use module_name
```

12 Symbol Aliasing

Aliasing allows you to assign an identifier as a stand-in for a symbol or constant expression. The scope an alias is the enclosing scope. An alias is a first class symbol, and can be referenced through the "." notation. Aliases are made using the binary infix ":=" operator. This operator creates an alias out of the name on the left hand side from the symbol or constant expression on the right hand side.

```
alias_symbol := existing_symbol_or_expression;
```

12.1 Importing Symbols from Other Namespaces

Aliasing is primarily a syntactic convenience. The most obivious use of aliasing is to import symbols from another namespace. For example, if you are using the "sin" function from the math module a lot, but do not which to use any other symbol in the math modules, you could do this:

```
require math;
sin := math.sin;
```

This effectively imports only the sin function into the current scope. Similarily if there is a module which is buryed deep within other namespaces, you can pull the module name into the current scope like this:

```
require some.very.far.away.module;
module := some.very.far.away.module;
module.call_some_function();
```

Aliases can also be assigned to other aliases. So for example:

```
require math;
sin := math.sin;
cos := sin;
tan := cos;

sin(123.321) == tan(123.321);   // eek! that's true!
```

This example "imports" the sin function into the current namespace and then maliciously calls it "cos" and "tan" as well. Chaos ensues.

12.2 Function Aliasing

You can alias function names. There are a couple instances where this becomes useful. The first is when a function name or a path to a function name becomes unweildy:

```
f := doSomeIncrediblyHeinousThingToAFloatingPointNumber;
float x = f(123.0);
```

In addition, you can use function aliasing to change a member function into a normal function:

```
append := float[].push_back;
float[] array;
append(array, 123.0);
```

In the above example, the push_back() member function is aliased to the name "append". The member function can then be called as a normal function. Note that the aliasing does not resolve virtual functions, it grabs the exact function specified on the right hand side. In addition, when a function name is overloaded, the alias represents all the overloaded functions. In effect, the alias is overloaded the same way the function name is. This is particularly evident in type aliasing; the alias not only represents a type name, but all constuctors for the type as well.

12.3 Type Aliasing

Perhaps the most usefull form of aliasing is type aliasing. This is essentially the same as the C language "typedef" statement. In Mu, to create a type alias you assign a type name to an identifier:

```
Scary := float[][100][50,123][][];
```

you can then use the "Scary" symbol in place of its alias:

```
Scary foo = Scary();
```

This can be particularily useful in cases where the meaning of array types becomes messy. Here's a very contrived example:

```
hpoint := float[4];     // a point is a homogeneous 4d object
cv := vector hpoint;    // a cv is a vector point
Patch := cv[4,4];       // a Patch is a 4x4 array of cvs
Surface := Patch[,];    // a Surface is resizable 2D array of Patches
Model := Surface[];     // A Model is a collection of Surfaces
Scene := Model[];       // A Scene is a collection of Models
```

When compiling, Mu will substitute symbols for their aliases recursively until a type is found. In the above case, the Scene alias expands out to:

```
Scene := (vector float[4])[4,4][,][][];
```

The standard modules use type aliasing. The math module declares some type aliases for vector types:

```
math.vec4f := vector float[4];
math.vec3f := vector float[3];
math.vec2f := vector float[2];
```

12.4 Variable Aliasing

Finally, a less useful form of aliasing is varible name aliasing. Again, the syntax is simple assignment to an identifier:

```
float foo = 123.321;
bar := foo;
print(bar + "\n");
```

Note the similarity to the folloing:

```
float foo = 123.321;
float bar = foo;
print(bar + "\n");
```

Both examples produce the same output, but syntactically two very different things are happening. In the first example, an alias to the variable foo is created called bar. Bar is not a new float variable; it is a second name for the variable foo. In the second example, an actual location in memory is created called bar and that second variable is assigned the value of foo.

12.5 Symbolic Constants

The symbol aliasing syntax can also be used to declare a symbolic constant. This is roughly equivalent to a macro in C. The symbolic constant will always reduce to a constant expression when used elsewhere.

```
pi := 3.14;
```

The symbol `pi` is not a variable and so cannot have its value changed.

13 Seperate Parse and Compilation Modules

13.1 Module Definition

13.2 File System Locations

13.3 Module as a Unit of Compilation

13.4 Different Flavors of Module

13.5 Loading Modules at Runtime

14 Documenting Source Code

Mu has built-in syntax for annotating source code symbols like functions, variables, and type definitions. When parsed and compiled the documentation is assigned to a symbol and can be retrieved at runtime along with other symbol information. (See Runtime Module).

Modules like `autodoc` can convert the documentation into various formats like plain ASCII, HTML, or a TEX dialect. (See Autodoc Module).

14.1 Source Code Comments

Mu uses C++ comment syntax

A double slash `//` comments to the end of the line.

A slash followed by a star `/*` begins a comment that may include newlines. The comment is terminated by `*/`.

14.2 Compiled Documentation

A documentation statement starts with the `documentation:` keyword followed by a string constant:

```
documenation: "documentation string goes here";
```

The parser will cache the documentation string until a symbol is defined in the same scope as the string. At the point, the parser will assign the documentation string to that symbol. If the scope changes, the documentation string may become orphaned.

An alternative more specific syntax makes it possible to specify the name of the next symbol to attach the string to:

```
documentation: foo "documentation string goes here";
```

In this case the next symbol defined as `foo` in the same scope as the documentation will be assigned the string. This makes it possible to string a number of documentation statements together before a compound definition (like a function) and document each symbol involved in the definition. This can be especially useful for documenting function parameters. For example:

```
documentation: rotate
"""The rotate function transforms a vector about the origin returning
the transformed vector.""";

documentation: axis
"""The axis about which to rotate""";

documentation: radians
"""The amount to rotate about the axis in radians""";

documentation: v
"""The vector to rotate""";

\: rotate (vector float[3];
           vector float[3] v,
           vector float[3] axis,
           vector float[3] radians)
{
    ..
}
```

14.3 Storage of Compiled Documentation

14.4 Documenation Syntax

15 Memory Management

Objects in Mu are automatically allocated and destroyed by the runtime environment. Objects created by the program, stack objects, and temporary objects in expressions are all handled in the same way. The Mu runtime environment uses a stanrdard mark-sweep garbage collection algorithm to find objects that are no longer being used and queues them up for finalization. When the objects are finalized, destructors are run and the object is reclaimed.

15.1 Allocation

There are three ways that objects are allocated: variable initialization, a constructor call, or as a temporary object during expression evaluation. When declaring a variable, the initialization either occurs directly or indirectly:

```
string foo = string(10); // foo = "10"
string bar; // bar = ""
```

By default, lack of an initialization expression causes Mu to supply the default constructor. In the above example, the "bar" variable is initialized like this:

```
string bar = string();
```

Object variables are never set to nil unless you explicitly do so:

```
string baz = nil;
```

The variable baz points to nothing. If you attempt to call a member function on baz, an exception will be thrown:

```
baz.size();     // throws! baz == nil
```

15.2 Deallocation

Mu uses garbage collection to reclaim unused memory. There are never dangling references in a Mu program. When a certain amount of time has passed, or a certain number of objects have been allocated, the runtime environment may invoke the garbage collector. Obviously, this has consequences on program design. You cannot rely on objects being finalized at a particular point in a program.

The garbage collector can be tuned from the runtime module:

```
require runtime;
runtime.collect(); // invoke garbage collector manually
runtime.set_collection_threshold(1000); // set object overhead parameter
```

16 Closures and Partial Evaluation

16.1 Closures

Closures serve two purposes in Mu. One is to wrap variable references in nested functions. The other is to wrap some function arguments to mimic partial evaluation.

When nested functions are used,

```
function: foo (int; int a, int b)
{
    function: bar (int; int c) { a * c; }
    bar(3.14) + b;
}
```

The `foo` function above will return `a * 3.14 + b`. The function `bar` is referring to a variable `a` in its scope. In essense, `bar` really has two arguments: `c` and another argument. Whenever `bar` is called, the extra argument's value is understood to be the value of the variable `a`. This is essentially syntactic sugar.

In this case, the closure is created whenever the function with hidden arguments is called. At the call point a closure object (which is a type of function object) is created that stores some of the parameter values. The closure object appears to be a function itself albeit with few arguments. The closure can be used anywhere a normal function object can be used.

In the second case, a closure can be directly created. For example let's say we defined a function that prints a message:

```
function: show_message (void; string msg)
{
    print("The message is: " + msg + "\n");
}
```

Now let's say there is a UI button object which will call a function with no arguments and no return value (type of `(void;)`). Since the `show_message` function has an argument of type string, it cannot be passed to the button:

```
button b = ...;
b.set_callback(show_message);   // error wrong type
```

However, let's say we just want a message to be printed when the button is pressed. We could do this:

```
function: adapter (void;)
{
    show_message("my message");
}

b.set_callback(adapter);    // ok
```

But we had to write an extra function. We could alternately use an anonymous function like this:

```
b.set_callback(\: (void;) { show_message("my_message"); });
```

but that's not too much better. Finally, we can create a closure around the `show_message` function by only partially applying it:

```
b.set_callback(closure(show_message, "my message")); // ok
```

In this case, the `closure()` function generated a closure object that appears as a function of type `(void;)`.

16.2 Constant Expressions

Mu evaluates expressions very aggressively. If the compiler/interpreter can find that an expression is constant, it will evaluate it immediately. This may occur before an expression is even finished parsing.

This is not unlike C/C++ compilers evaluating constant expressions at compile time.

In addition, all functions are tagged with information about possible side effects or lack thereof. Because of this, even functions in modules can be evaluated early in order to fold constant values. Here's an example:

```
float f = math.acos(math.cos(math.pi) * -1.0);
```

the interpreter reports that this evaluates to the runtime expression:

```
float f = 0.0;
```

During parsing the interpreter determined that `math.cos` and `math.acos` are side effect free and therefor a constant argument will produce a constant return value. This is one form of partial evaluation.

16.3 Explicit Partial Application

Functions may be partially evaluated (applied) explicitly by using the empty argument syntax. This only applies to functions which take more than one argument.

The result of a partial function evaluation is a new function which will have the same return type as the partially evaluated function but a subset of its arguments. The following interactive session provides a basic example:

```
mu> \: add (int; int a, int b) { a + b }
mu> add(1,);  // NOTE: missing arg
(int;int) => __lambda (int; int) (+ 1 __lambda.a)
mu> add(1,)(10);
float => 11.0f
mu> add(1,10)
float => 11.0f
```

Notice that when you call the function `add` with all its arguments it does what you expect: returns an `int`. When you call with *some* of its arguments it returns a new function.

16.4 Explicit Partial Evaluation

17 Phases

Mu has three phases: parse, compilation, and runtime evaluation.

17.1 Parse Phase

17.2 Compilation Phase

17.3 Runtime Phase

18 Muc: Mu Compiler

18.1 Invocation

18.2 Options

18.3 Muc Target

18.4 C++ Target

19 Standard Library

19.1 Built-in Functions

Printing Values String Formating size

19.2 Runtime Module

19.3 Autodoc Module

19.4 Math Modules

19.5 I/O Module

serialize

19.6 OpenGL Related Modules

19.7 POSIX Functions

19.8 Image I/O, Storage, and Operations

19.9 OpenCV Functions and Data Types

19.10 Read/Write GTO Files

20 Mu Compared to Similar Languages

20.1 C++

20.2 Python

20.3 JavaScript

20.4 ML Family of Languages

21 Example Usage

21.1 Embedding Mu in a C++ Application

21.2 Using Mu as a Shading Language

21.3 Using Mu to Control Particle Dynamics

21.4 Mu by Itself

Appendix A Reference

Properties

(Index is nonexistent)

Functions

A

C

H

T

Types

(Index is nonexistent)